# On Hardware-Agnostic Programming for Parallel Architectures

Thomas Koopman

Institute for Computing and Information Sciences

RADBOUD UNIVERSITY PRESS

Radboud Dissertation Series

# On Hardware-Agnostic Programming for Parallel Architectures

Thomas Johan Koopman

**On Hardware-Agnostic Programming for Parallel Architectures**
Thomas Johan Koopman

**Radboud Dissertation Series**
ISSN: 2950-2772 (Online); 2950-2780 (Print)

Published by RADBOUD UNIVERSITY PRESS
Postbus 9100, 6500 HA Nijmegen, The Netherlands
www.radbouduniversitypress.nl

Design: Thomas Johan Koopman
Cover: Proefschrift AIO | Guntra Laivacuma
Printing: DPN Rikken/Pumbo

ISBN: 9789465151786
DOI: 10.54195/9789465151786
Free download at: https://doi.org/10.54195/9789465151786



RADBOUD
UNIVERSITY
PRESS

# On Hardware-Agnostic Programming for Parallel Architectures

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

donderdag 7 mei 2026
om 12:30 uur precies

door

Thomas Johan Koopman
geboren op 6 juli 1998
te Leidschendam

Promotor:

prof. dr. S.-B. Scholz

Copromotoren:

dr. P.M. Achten
dr. P.W.M. Koopman
dr. B.E. van Gastel

Manuscriptcommissie:

dr. R.J. Krebbers
prof. dr. A.-L. Varbanescu, Universiteit Twente
prof. dr. R. van Nieuwpoort, Universiteit Leiden

# Contents

# Chapter 1

# Introduction

Choosing a programming language makes a trade-off between abstraction and control. Before 1957 most scientific codes were programmed in assembly code directly for the instruction set of the target machine. This exposes all capabilities of the machine, but limits the code to machines with that instruction set. Different versions of the same code are necessary to support multiple instruction sets. Moreover, writing these assembly codes is error-prone and time consuming, which can make it harder to improve the high-level structure. The introduction of FORTRAN, one of the first high-level languages, has changed this by abstracting the instruction set away, making it possible to use the same code on any machine with a FORTRAN compiler. FORTRAN did not only increase the portability of programs, but also made them easier to write. Though the loss of control over the machine instructions can inhibit the performance of programs, FORTRAN and other higher-level languages are far more widely used than hand-written assembly today.

FORTRAN is designed for scientific computing, i.e. computing mathematical functions rather than applications like word processing, websites, or databases. This is an interesting area of computing because these applications are computationally expensive. Despite the name, scientific computing is not restricted to research centres. For example, playing media files and telecommunication use the same computational core as some climate simulations. Video games and animated motion pictures spend much of their processing time on solving an equation describing how light interacts with objects. The training of large language models is essentially about finding the minimum of a mathematical function.

Today, different instruction sets are just the tip of the iceberg. In the past, processors increased their performance by increasing the clock speed. This progress has stalled around 2005 due to heat dissipation issues [180] and other problems. Instead, chip manufacturers now increase performance by combining processors. The idea is to split up a task in smaller tasks that can be computed in parallel by the processors. If we can split up the task and properly coordinate the smaller tasks, then the total computation will finish faster. There are many ways to create these parallel systems, which has led to a wide range of parallel architectures,

such as multicore CPUs, GPUs, FPGAs, and combinations of these in distributed systems. The different processors and memories have to be coordinated to collaboratively compute a task. These processors each come with their own programming environment, such as OpenMP [189], CUDA [182], or MPI [241] and many more.

Typically, an algorithm is implemented separately in each of these programming environments. Hence, we have returned to a situation similar to the situation before the introduction of FORTRAN: we need multiple source codes, each of which is written specifically for a particular architecture. Again, the same goal of generating performant code from a single source code for a variety of architectures becomes relevant. This raises the question: can we abstract over parallel architectures the way FORTRAN abstracted over instruction sets? Doing so is important for the same reasons, so I pose the following thesis: *it is time to abstract away hardware architecture in scientific computing.* FORTRAN has not made assembly code completely obsolete, and we do not expect hardware-specific code to become completely obsolete either. However, we delay nuances to the conclusion as they are too technical and specific to discuss in the introduction.

As abstracting over hardware was done successfully—in a time of simpler hardware—by FORTRAN, we should aim to understand why. Two reasons come to mind. First, many abstractions introduced by FORTRAN are shallow due to the similarity in processors at the time. For example, all processors had instructions for adding two numbers, so it is easy to form the abstraction +, and to generate the correct assembly instruction from this. For the second reason, we should distinguish between two types of optimisations, which I will call local and global. Local optimisations are done on small sequences of instructions that can be studied in isolation, like a loop body. For example, we may have a sequence of instructions that contain an addition and a multiplication that can be executed in any order. We would like to do the multiplication first on an architecture with a larger multiplication-latency than addition-latency, and we would like to do the addition first on an architecture where this is reversed. These local optimisations are easy to automate, and a compiler is usually better at it than human programmers. In contrast, global optimisations affect the overall program structure, for example choosing a more suitable data structure. These optimisations are easier to do by humans in a readable language such as FORTRAN, than in an assembly language. Only local optimisations require knowledge of the architecture, and as these optimisations are automated by the compiler, it is not necessary to expose them in the language. Therefore, FORTRAN can achieve high performance while abstracting over the instruction set.

The idea to abstract away hardware in the era of parallel computing is not new. Since the shift towards increasingly parallel hardware, many approaches have been developed. One popular approach is to write libraries for all important functionality, for example BLAS [30], LAPACK [8], PETSc [17], and TensorFlow [2]. This is easy for application writers, who can simply glue these calls together in a language such as Python. However, as soon as no library is available, the performance will suffer severely. Furthermore, there exists a virtually endless supply of functionality, and it is infeasible to write hand-optimised versions for all these functions, for all common hardware.

Another approach is to write a library that exposes programming patterns rather than specific functionality. For example, Kokkos [230] and Thrust [111] are general-purpose C++ libraries that abstract over different hardware targets. This approach makes it easy to create well-tuned implementations for library functions, but is limited in optimising between library calls, as it relies on a C++ compiler that is unaware of the library's high-level semantics.

Algorithmic variants have a large effect on the performance of a program. The desire to optimise in this space, motivates languages that are aware of algorithmic variants in a problem domain. These are called *domain-specific languages* or DSLs. DSLs are used in signal processing (FFTW [82], SPIRAL [200]), image processing (Halide [201]), and solving partial differential equations (UFL [6]). A DSL approach is not mutually exclusive to using a library or a general-purpose language. For example, the Firedrake [100] framework compiles UFL to C-code that makes heavy use of the PETSc library. Furthermore, a compiler could translate DSL to a hardware-agnostic language. The cost of using a DSL is duplicating effort for every problem domain, as optimisations that are not domain-specific have to be reimplemented for each DSL.

General-purpose, hardware-agnostic languages avoid the duplication of effort in both problem domain and hardware target. Let us consider what constructs such a language should have from a minimalist angle. What low-level features would a language have to take away, and what language constructs are useful to add? Different parallel architectures have different memories and different types of execution units, so a language that abstracts over parallel architectures has to take away at least memory, and explicit mappings of computations to execution units. This makes the compiler responsible for parallelising the code, so we need it to identify independent chunks of work. For this reason, it is useful to add arrays to the language: operations on arrays consist of many independent operations on elements. However, arrays alone are not enough. If we call a function on each element in an array that also prints the element to a terminal, we cannot execute it in parallel, at least not without accepting an arbitrary order. In general, it is difficult to analyse how functions change global state. Functional languages make global state explicit, allowing them to isolate it to well-defined language constructs. For this reason, all general-purpose and hardware-agnostic languages that we are aware of, are functional array languages. For example, SISAL [74], Accelerate [169], Single-assignment C [96] (SaC), Futhark [104], and APL [117] are functional array languages. It may seem impractical to build a new language from scratch, but it is interesting to note that the codebase of Kokkos library is similar in size to the compilers of these languages. So while there is a high initial cost to implementing a new language, the additional cost of adding functionality, optimisations, and hardware targets can be smaller than a library approach.

So far, we have considered parallel programming as a combination of computational resources. This combination can be achieved by using interconnects to pool the resources of different processors. We call the result a *distributed-memory* system because processors cannot directly access the memory of other processors. This adds the challenge of using the combined memory efficiently. An algorithm or system that uses the combined memory efficiently is called *memory-scalable*.

A minimal abstraction over distributed-memory systems is to program from the perspective of a single process, but to abstract the inter-process communication, such as provided by MPI. This approach makes the programmer responsible for distributing the data over processors: the correspondence between the global data structure and collection of local data structures exists in the head of the programmer, or in a data structure made by the programmer. Languages such as Chapel [43], X10 [49], and UPC [86] increase the level of abstraction by adding these data structures to the language. This provides some support for the programmer, but data distributions are still explicit. We do not want to expose this hardware-specific detail, but no languages with fully implicit parallelism—such as SaC, Accelerate, APL, and Futhark—generate memory-scalable code. SaC does target distributed-memory systems [162], so the first contribution of this dissertation is to build on this, and generate memory-scalable code among other improvements. Or in other words, to improve local optimisations made by these languages for distributed-memory architectures.

Higher-level languages offer productivity and portability at the cost of control. When trying to understand FORTRAN's success, we hypothesised that this is only a problem if it inhibits us from making optimisations in the global structure. So while it is possible to obtain high-performance by encoding low-level algorithmic optimisations in languages (for example tiling [244] in Halide), we argue a different approach would also lead to good performance, without complicating the language with low-level details. The second contribution of this dissertation is to find new ways to employ an existing language feature for global optimisations. To this end, we investigate a defining feature of Single-assignment C called *rank-polymorphism*: the ability to process arrays independently of their rank (also called dimension). Related work uses rank-polymorphism to control parallelism [220], and we extend this use to other areas.

The two-part structure of this dissertation reflects these two contributions. In Part I 'Hardware-independent Optimisations', we investigate to what extent it is possible to generate code for different hardware architectures from general-purpose languages without explicit parallelism or manual memory management. In Subparts: Subpart A 'Case Studies', we evaluate existing approaches on CPUs and GPUs. In Subpart B 'Code Generation', we address the research gap in generating memory-scalable code.

Chapter 2 is at the centre of Subpart A, and aims to generate performant code for both CPUs and GPUs from a single source code in different functional array languages. This is largely successful, but a benchmark called Quickhull is out of reach for current compiler technology. To show this is an engineering—rather than conceptual—challenge, we approach the core part of Quickhull from a data-movement perspective that is inspired by distributed computing, but independent from hardware. We then translate this technique to efficient implementations for both GPUs (Chapter 3) and CPUs (Chapter 4) by hand. This can serve as a starting point for code generation.

Subpart B aims to extend and improve support for distributed-memory code generation. These systems are typically programmed manually by explicitly moving data between memories. This is unfeasible for code generation because we do

not have enough guarantees to statically decide the location of data. Checking where data is stored on each access is prohibitively expensive. We approach this problem by mimicking a shared memory system. This idea is called distributed shared-memory [157] and typically also prohibitively expensive. However, generating code from an array language gives us guarantees that we can exploit. In Chapter 5, we propose our own variant that takes this into account, and implement a library that is suitable as a compilation target for array languages. Chapter 6 then shows how to generate this code from the array language Single-assignment C. We can do something similar for systems with multiple GPUs, where we use the vendor-supplied distributed shared memory system (Chapter 7).

In Part II 'Rank-polymorphic Algorithm Design', we explore three new uses of rank-polymorphism. In Chapter 10, we show how the essential idea behind optimised matrix-multiplication can be clearly expressed using a function that operates on arrays of arbitrary rank. So for this widely used computational kernel, optimising the global structure is sufficient to obtain state-of-the-art performance. From a more theoretical perspective, it turns out that for a specific class of parallel algorithms, we can generalise a one-dimensional algorithm to higher dimensions, without changing the algorithmic structure (Chapter 8). This can be applied to obtain a novel rank-polymorphic algorithm that we extensively benchmark in Chapter 9. Finally, we show that using the rank of an array to guide computation cannot only help performance engineering, but can also help to minimise numerical error. In Chapter 11, we use the same idea from matrix-multiplication to reduce the error bound in finding the point that is farthest from a line.

## 1.1 Statement of Contributions

The chapters of this work are based on peer-reviewed papers, or papers that are currently under peer review (except for Chapter 6). I will clarify my part in creating these works, and differences between the chapters and papers. For Chapters 3, 4, 6, 8, 9 and 11, I was the main author and responsible for the central research idea, software, experimentation, and initial draft. My co-authors supervised the research process, provided feedback, and helped with the writing. For the other papers, I will clarify my contributions in the enumeration.

Chapter 2

D. van Balen, T. De Matteis, C. Grelck, T. Henriksen, A.W. Hsu, G.K. Keller, T.J. Koopman, T.L. McDonell, C. Oancea, S.-B. Scholz, A. Sinkarovs, T. Smeding, P. Trinder, I.G. de Wolff, and A.N. Ziogas. *Comparing Parallel Functional Array Languages: Programming and Performance.* 2025. arXiv: `2505.08906 [cs.PL]`. URL: `https://arxiv.org/abs/2505.08906`

Under review.

Only the language implementation and benchmark sections have been included in this work to keep its length manageable.

I implemented and measured the SaC benchmarks. I also implemented and measured the CPU baseline for the N-body and Flash

Attention benchmarks.

Chapter 3 T.J. Koopman, S.-B. Scholz, and B. van Gastel. "Partitioning In-Place on Massively Parallel Architectures". In: (2025). DOI: 0.1007/978-3-031-99872-0_7

Chapter 4 T.J. Koopman, J. Aaldering, B. van Gastel, S.-B. Scholz. *"VQhull: a Fast Planar Quickhull"*. Under review.

I designed and implemented the sequential algorithm and parallel step of the subset extraction. Jordy Aaldering designed and implemented the cleanup phase. Experimental evaluation was a collaborative effort with Jordy Aaldering.

Chapter 5 S. Schrijvers, T. Koopman, and S.-B. Scholz. "Shray: An Owner-Compute Distributed Shared-Memory System". In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2024. DOI: 10.1145/3652586.3663314

The design was a collaborative effort with Sven-Bodo Scholz and Stefan Schrijvers. Implementation and evaluation was a collaborative effort with Stefan Schrijvers.

Chapter 6 T.J. Koopman and S.-B. Scholz. *"Generating Distributed Code from Array Languages"*.

Chapter 7 P. van Beurden, T.J. Koopman, and S.-B. Scholz. "Multi-GPU Code Generation for Out-of-Core Problems". In: *Trends in Functional Programming*. 2025. DOI: 10.1007/978-3-031-99751-8_6

I shared some insights of my work on Chapter 6 during the design, implemented the compiler phase deciding whether to give memory advice, and worked on the experimental evaluation.

Chapter 8 T.J. Koopman, Rob H. Bisseling, and S.-B. Scholz. *"Category Theory for Supercomputing: The Tensor Product of Linear BSP Algorithms"*. Under review.

Chapter 9 T.J. Koopman and R.H. Bisseling. "Minimizing Communication in the Multidimensional FFT". in: *SIAM Journal on Scientific Computing* 45.6 (2023). DOI: 10.1137/22M1487242

Chapter 10 A. Šinkarovs, T.J. Koopman, and S.-B. Scholz. "Rank-Polymorphism for Shape-Guided Blocking". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. 2023. DOI: 10.1145/3609024.3609410. The formal verification part has been stripped out. Except for Section 10.4, the chapter has been modified to reflect this.

The idea of using shape for blocking came from a discussion with Sven-Bodo Scholz. The implementation was a collaborative effort with Artjoms Šinkarovs. I did the experimental evaluation.

Chapter 11 T.J. Koopman and S.-B. Scholz. *"Quickhull is Usually Forward Stable"*. Under review. The note on correctness and shapes (Section 11.8) has not been submitted.

**Work not included in this thesis.**

- M. Verloop, T.J. Koopman, and S.-B. Scholz. "Modulo in high-performance code: strength reduction for modulo-based array indexing in loops". In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. 2024. DOI: 10.1145/3652561.3652573

  I provided correctness proofs for the number-theoretic claims.

# Part I

# Hardware-independent Optimisations

# Subpart A

# Case Studies

# Chapter 2

# Comparing Functional Array Languages

## 2.1 Introduction

Parallel programming is hard. Not only must the developer solve the challenges of sequential software, i.e. specify a correct and efficient algorithm operating over appropriate data structures, but they must also specify parallel coordination, e.g. how the computation is broken into sub-computations, how the sub-computations communicate and synchronize, how they are mapped onto cores, etc. Some parallel hardware is notoriously hard to exploit. For example, obtaining good performance on a GPU requires careful mapping of computations to the hardware, and careful management of the memory hierarchy, e.g., transferring data between GPU and CPU memories or effectively utilizing scratchpad memory as a cache.

The variety of parallel architectures, and their rapid evolution, require performance portability. Parallel language designers aim to generate efficient code for multiple architectures, e.g., enabling a program to be compiled for both multicores and GPUs. If a new generation of hardware increases the number of cores or the amount of memory, the parallel program may need to be refactored for the new architecture. A key objective is to design languages that minimize the refactoring required for a new architecture.

One means of taming the challenges of parallel programming is to focus on some important class of computations. Array languages like APL [117], Fortran [15], or R [118] are designed to express computations on arrays concisely and to implement them efficiently. These languages one way or another generalize operations on scalars to apply transparently to vectors, matrices, and higher-dimensional arrays. Language implementations are carefully designed to exploit the memory hierarchy and often use hardware capabilities such as vector units.

Some array languages are functional, e.g. APL is a design that has stood the test of time. More recently functional array languages like Accelerate [169], DaCe [252], Futhark [104] and SaC [96] have emerged. These languages offer the

attractive prospect of low-effort parallel array programming, good performance, and performance portability between architectures.

Although functional array languages adopt a range of design, implementation, and optimisation decisions, they have some core commonalities. They use a data-parallel model where bulk parallel operations are performed over the arrays, e.g., mapping a function over every element of an array. This model greatly simplifies parallelisation as it avoids explicit synchronisation and communication. Parallel functional array languages further avoid typical issues of parallelism like race conditions or deadlocks and lifelocks strictly by design. However, key questions remain. Are such high-level parallel programming models adequately expressive? What are the implications of functional array language design choices? Can the language implementations deliver performance competitive with conventional languages?

We investigate these questions by comparing the designs, implementations, and performance of the five parallel functional array languages Accelerate, APL, DaCe, Futhark, and SaC.

We make the following research contributions:

- We systematically compare the implementations of the five functional array languages. Key implementation aspects are the implementation model (e.g., compiler or DSL), the target architectures, and the supported optimisations.

- We show the expressiveness of functional array programming using four challenging benchmarks, namely N-body simulation, MultiGrid, Quickhull and Flash Attention (Sections 2.4 to 2.7). These benchmarks represent a range of application domains and parallel computational models and require different optimisations. Rather than choosing simple benchmarks that can easily be expressed and effectively be optimized in all five languages, these challenging benchmarks often take the languages outside of their "comfort zone". Crucially, we argue that *the functional array code is more comprehensible than the hand-optimized baseline implementations*(Section 2.8).

- We compare the codebase sizes, and hence approximate the programming effort, of the benchmarks in the functional array languages and in the OpenMP and CUDA baselines. We use Source Lines Of Code (SLOC) as a crude but widely accepted measure [204]. A major difference is that the functional array programmer writes a single program that is compiled for multiple targets, where there are separate CPU and GPU baselines. Hence *the total baseline codebase is much larger, at least* $10\times$ *than any of the functional array codebases. The functional codebases are at least* $2\times$ *smaller than the CPU baseline, and at least* $8\times$ *smaller than the GPU baseline codebases* (Section 2.8).

- We investigate to what extent performant implementations can be generated from a single high-level source for multiple architectures. Can performant code be generated even although the functional array code omits important architecture-specific information, e.g., aspects of the memory hierarchy? We

find only partial evidence as only Dace and Futhark consistently achieve good performance on both multicore and GPU (Section 2.9).

- We report benchmarking results for the five languages on both a 32-core AMD EPYC 7313 multicore system and on an NVIDIA A30 GPU (Sections 2.4 to 2.7). We analyze the multicore and the GPU performance of the languages on 39 instances of the four open-source benchmarks to explore why each language performs well, or poorly, on each benchmark and architecture. We find that in 30 % of the instances, at least one language matches or outperforms the baseline; that in 70 % of the instances, at least one language achieves more than 80 % of the baseline performance. In only 6 % of instances, no language achieves more than 50 % of the baseline performance. *We argue that the results show that mature functional array languages have the potential to deliver performance competitive with the best available conventional techniques* (Section 2.8).

## 2.2 Language Implementations and Comparison

This section outlines the implementation of the five functional array languages, before making a systematic comparison of languages covering implementation model, execution targets, and key optimisations supported.

### Implementing the Futhark Language

The Futhark compiler is a heavily optimizing, but architecturally conventional, ahead-of-time compiler. It generates code for both GPUs (targeting OpenCL, CUDA, or HIP) or CPUs (targeting POSIX Threads). Each compilation target has a distinct optimisation pipeline, although with significant overlap in passes. Compilation begins by compiling away module-level constructs, monomorphizing polymorphic functions, defunctionalizing higher-order functions, and flattening compound types, such as tuples. Arrays of compound types are represented as *multiple* arrays, each storing only primitive scalar types. For example, a source array of type [n](i32,bool) is represented in the compiler as two arrays of type [n]i32 and [n]bool. This is generally known as a structure-of-arrays (SoA) representation. Hence the remainder of the compilation operates on a first-order, monomorphic program with only primitive types and arrays of primitives, and a handful of built-in second-order array combinators (SOACs) such as maps, scans (prefix sum), reductions, and generalized histograms [101].

The compiler performs standard optimisations, such as copy propagation, constant folding, inlining, common subexpression elimination, loop hoisting, and so on. The most important early optimisation is ***fusion*** (or *deforestation*) [240], where adjacent array traversals are combined to avoid materializing intermediate arrays. Many fusion rules are used, and a simple example fuses maps, i.e. replaces traversals for f and for g with a single traversal for the composition f ∘ g:

$$\text{map f} \circ \text{map g} \Rightarrow \text{map (f} \circ \text{g)}$$

Futhark supports nested parallelism, but some compilation targets, like GPUs, only efficiently support a fixed number of parallel levels, and there are significant performance implications in how they are used. Therefore, one of the most important transformations is *flattening* [33], where multiple levels of application-level parallelism are collapsed and assigned to different hardware levels. This is achieved by ***incremental flattening*** [105], which uses map fission and loop interchange to create semantically equivalent code versions that utilize more and more levels of application parallelism. Essentially, using a top-down program traversal, whenever a new map f operation is discovered, the analysis:

(1) creates a first version corresponding to a CUDA kernel in which each thread (independently) executes an application of $f$.

(2) creates a second version, dubbed *intra-block kernel*, in which the map parallelism discovered so far is mapped on the CUDA grid, and the parallelism inside $f$ is flattened and mapped to the CUDA block level; this has the benefit that intermediate arrays are created and reused from fast memory.

(3) continues flattening recursively (by means of map fission and loop interchange) in a parallel context that is extended with the current map.

The resulting code versions are combined together into a program that branches depending on some dynamic program measure with a threshold. The best combination of code versions is derived by autotuning the threshold values [177].

Subsequent optimisation passes (i) rewrite code using accumulators in terms of more specialized constructs such as reduce(-by-index) [206], (ii) optimize temporal locality, e.g., tiling in registers and shared memory is performed when an array that is invariant to one of the parallel dimensions is "streamed" through an operator, and (iii) minimize the footprint of scratchpad memory and eliminating unnecessary copy operations [178].

Finally, SOACs benefit from specialized GPU code generation: e.g., the implementation of scan [183, 54] uses Merrill and Garland's single-pass algorithm [173], and that of reduce-by-index uses a combination of multi-pass and multi-histogram techniques to improve locality and reduce atomic conflicts [101].

## Implementing the Accelerate DSL

As Accelerate is a deeply embedded language constructs do not directly operate on arrays or scalar values. Instead, constructs operate on abstract syntax trees (ASTs) that represent computations of arrays and scalars. Users are mostly shielded from this fact, and write programs that look like regular Haskell programs. An embedded implementation brings a number of advantages: much of the mature host language infrastructure, such as the GHC compiler front-end, run-time system, and some of the memory management can be used. However, embedding also means that generating the code incurs runtime overhead. The implementation aims to minimize the overhead by optimizing compilation time and caching compiled code so that if the same code is run multiple times, it is only compiled once.

Accelerate offers currently three main, fully implemented back-ends: one that targets NVidia GPUs generating PTX code via LLVM, a multicore CPU back-end (also via LLVM), and a sequential interpreter for debugging. However, most of the implementation is platform-independent.

As for Futhark, and many other languages where array operations are expressed as higher-order functions, fusion is a core optimisation [169]. Similarly, regular nested operations are flattened, and the necessary monomorphic instances of polymorphic operations generated. Scans and folds in inner loops are only executed in parallel if the range of the parallel outer loop is too small to generate sufficient parallelism. The mapping of convenient surface types (which the user programs with), such as arrays of compound types, to machine-friendly arrays of primitive types is also similar to Futhark and other array languages in that it results in a SoA representation. In Accelerate, the transformation to SoA form is implemented as part of the translation from surface to internal types, and exploits GHC's type families and associated types [213, 46].

In addition to language optimisations, Accelerate requires some additional optimisation specific to embedded languages. One such optimisation is *sharing recovery* that maps user-friendly higher-order embedding with lambda abstractions and let-bindings, into a first-order embedding [169]. Runtime compilation also allows for additional specialisations of the code.

## Implementing the Single Assignment C Language

Although the name and syntax of Single Assignment C may suggest a light-weight implementation on top of a C compiler, the reality is very different. The sac2c SaC compiler is a many-pass compiler for a language that is syntactically very similar, and often identical, to C; Fig. 2.1 provides an overview.

The first block of code transformations is ***functionalisation*** where the intermediate code is transformed from the imperative-flavored source SaC into a representation that makes the underlying functional semantics more explicit. For example, loops are transformed into tail-recursive functions, if-then-else constructs into conditional expressions, etc.

The next block of compiler passes performs type inference and checking. In contrast to C, variables in SaC can be introduced on the fly and without declaring their types.

The high-level optimisation pass is crucial, and most optimisations are architecture-independent. Figure 2.1 also enumerates the most important optimisations. The optimisations include both conventional compiler optimisations, like copy propagation, and specific optimisations for functional array processing, like the various with-loop optimisations listed in Fig. 2.1. The SaC compiler sac2c aims at fusing with-loops in three dimensions: Vertically, it fuses with-loops where the result of one becomes the input to another; we call this with-loop-folding. Horizontally, it fuses (or tuples) with-loops that compute results based on the same or at least overlapping set of argument arrays; we call this with-loop-fusion. And in the remaining dimension, we fuse nested with-loops where one computes the element-wise value of enother; we call this with-loop-scalarisation. With-loops

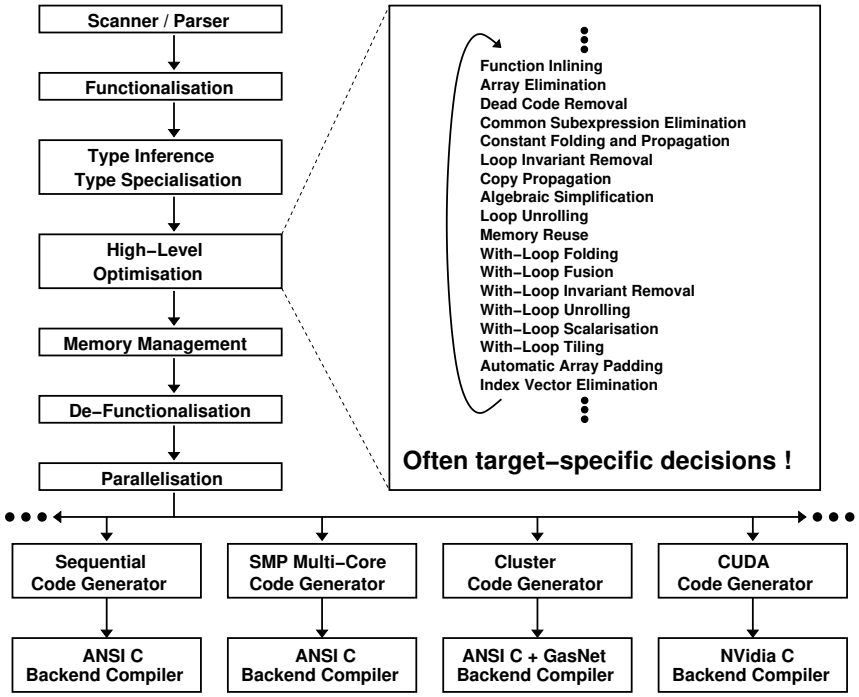



Figure 2.1: Structural organisation of the SaC compiler sac2c

are SaC's one-stop-shop solution to both specify and internally represent data-parallel array operations; for details see [95]. Any language-level array operation is converted into some with-loop. All optimisations benefit from the underlying purely functional semantics of SaC, and can usually be applied more aggressively than with an imperative interpretation of syntactically identical or very similar C code.

The memory management pass associates abstract arrays with concrete memory. This is where we control memory reuse and go to great lengths to counteract the aggregate update problem, both key aspects to achieving high performance with functional arrays. Details can be found in [97].

The next pass *defunctionalizes* the IR for the imperative target architectures. For example, tail-recursive functions are converted back into loops, etc.

Parallelisation is a surprisingly small pass, largely because the functional array code is typically highly parallel in nature, and all array operations are internally represented by a single IR construct: the with-loop. With-loops typically have more concurrency than most architectures have execution units. For this reason, the compiler focuses on how much work to map to each processor (be it a core, a node, or a thread).

In a final pass, the SaC compiler selects and parameterizes the backend compiler

to produce a linked executable. A more detailed description of the SaC compilation pipeline is available in [93].

## Implementing the APL Language

APL has multiple implementations, and in the benchmark sections we use the Dyalog interpreter on the multicores and the Co-dfns compiler on the GPUS. Dyalog is a bytecode interpreter for APL source token streams implemented in C/C++. The interpreter calls a specialized function for each APL primitive. Most primitives have multiple implementations designed to run more effectively on different input shapes, data ranges, etc. For some primitives, a minor degree of multi-threading is used if the data ranges are large enough, but this appears to have little impact on the benchmarks used here.

The design and architecture of the Co-dfns compiler is novel [114], and based on an extension of the NanoPass architecture [132]. It consists entirely of small, incremental, data-parallel passes written in APL to construct the entire compiler pipeline. It uses almost zero abstractions over pure APL arrays, where the AST is represented using an SoA design and is exceptionally efficient in its data representation. Outside of the major pipeline functions encapsulating the parser, compilation, and code generation phases, it contains almost no function abstraction nor many internal temporary variables, but only primitive APL function expressions over a few AST fields and temporary variables. The code, thus, lacks explicit looping constructs, branching, conditionals, or other explicit control flow outside of the use of the power operator to create iteration to fixed points for certain passes. As a result, the compiler is exceptionally compact, while being data-parallel by construction and thus GPU compatible.

The Co-dfns compiler optimizes very little at compile time. Efficient data representations, fusion of APL primitive calls, GPU acceleration, inlining of functions, etc., all utilize runtime implementations or the backend compiler, such as a C compiler. Fusing primitive calls in order to reduce the materialisation of intermediate arrays and to reduce the number of GPU kernels generated is accomplished using existing JIT and GPU libraries. [165] This simplifies the compiler while at the same time enables features such as fusion to occur across user-defined function call boundaries. However, the computations being fused must be combinations of pre-existing APL primitives. For example, reductions using max, min, or plus can all be fused with scalar primitives in a single pipeline, but reduction with a user-defined function cannot be fused. The runtime can detect the use of some known functions, and if idiomatic combinations are used, such as $+.\times$ for matrix multiplication, the runtime dispatches to an optimized library function specialized based on a combination of the datatypes involved and the functions being called. However, these specialisations are limited to what the runtime can see. Thus, optimisations like algebraic-style expression simplification are not applied.

The latest version of the Co-dfns compiler uses a runtime implemented in APL, with a small subset of functionality provided by the underlying host platform, such as C or JavaScript, in the form of a small kernel library. This is achieved using a language-agnostic foreign function interface.

## Implementing the DaCe Framework

DaCe expresses program optimisations as graph transformations on the SDFG IR [21]. Transformations are applied in optimisation passes, and may alter any aspect of the graph representation. They may be applied to the whole program, or only to specified subgraphs. Multiple passes can be combined into *pipelines*, ensuring that pass dependencies are met. DaCe provides a large library of transformations, passes, and pipelines. The framework includes user-level APIs to allow the development of new transformations and pipelines [250, 22]. Although many of the built-in passes and pipelines can be applied automatically, the user can opt to optimize a DaCe program manually. Manual optimisation can use a set of APIs or visual tools , that guide minimizing data movement [205]. DaCe also provides APIs for instrumenting, optimizing, and auto-tuning SDFG programs through performance modeling with built-in or external tools [251, 231].

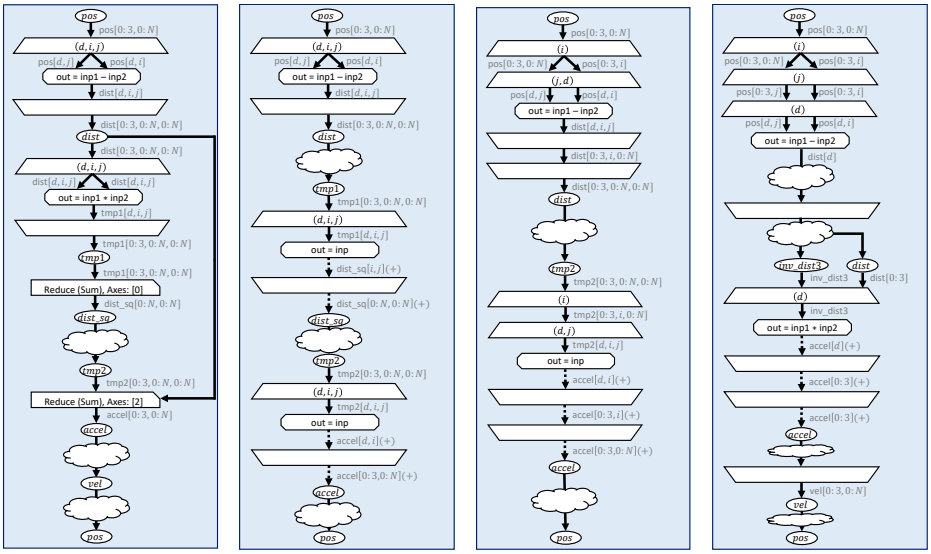



Figure 2.2: Optimisation workflow for the acceleration computation of the $N$-body program in DaCe: (a) unoptimized SDFG IR, (b) Library Nodes' expansion, (c) Map scope dimensions' permutation and *MapExpansion*, (d) *SubgraphFusion*.

We illustrate some built-in data-centric transformations for the $N$-body program, focusing on the single iteration of the algorithm shown in Fig. 2.2a. The first optimisation expands the Reduce Library Nodes to Map scopes, and must introduce Write Conflict Resolution (WCR) on the output edges, as multiple Map iterations write to the same memory location (Fig. 2.2b). Next, we permute the dimensions of selected Map scopes and apply the *MapExpansion* transformation so that all subgraphs have an outer iteration over the bodies using index $i$ (Fig. 2.2c). Finally, we apply the *SubgraphFusion* transformation to fuse all subgraphs up to

(and including) the computation of the updated velocities (vel) under a common outer Map scope (Fig. 2.2d). The transformation does not apply to the Map scope that updates the bodies' position, as that would produce read/write conflicts on pos. Fusing the Map scopes renders all intermediate results thread-local and reduces their dimensionality. For example, dist and accel become three-element vectors, and inv_dist3 reduces to a scalar value.

Specialisation for a specific architecture like a GPU also occurs in the SDFG IR. Data containers have storage types, and Maps have schedule types. For example, to "convert" the $N$-body program for execution on a GPU, we simply switch the program's inputs and outputs to *GPU-global* memory. Similarly, the Maps' schedules must be changed to *GPU-device* and *GPU-thread-block*. Storage and schedule changes can be applied manually (programmatically or using visual tools) or automatically by applying device-specific transformations, such as *GPU-Transform*, which iterates over the SDFG and updates the schedule of Map scopes deemed suitable for GPU execution. Data movement between the host and device is explicit in the SDFG and can also be automated with the same device-specific transformations.

Since a program's SDFG representation includes scheduling and storage specification, DaCe's code generation is relatively simple. The first step is to traverse the SDFG to determine the lifetime of all transient variables, marking the appropriate states for allocation and de-allocation. Subsequently, DaCe partitions the SDFG into subgraphs defining control-flow scopes. For loops and structured control, DaCe generates the corresponding C/C++ syntax, e.g., for, while, if, etc. Unstructured control flow is generated as a state machine with gotos. The body of the control flow scopes is generated state-by-state in a topological order. Each state is code-generated in three steps: (1) transient allocation, (2) dataflow code generation, and (3) transient de-allocation. To generate code for dataflow, DaCe traverses the graph in topological order, generating appropriate code based on the schedule and storage types, e.g., a map with a *CPU-multicore* schedule generates an OpenMP for-loop, where a GPU-related schedule generates a CUDA/HIP GPU kernel.

## Language Implementation Comparison

This section compares the implementations of the five functional array languages. There are a variety of implementation techniques: the Futhark and SaC compilers, the Accelerate DSL, the DaCe framework, and the APL interpreter/compiler. The key differences are summarized in Table 2.1.

The SaC and Futhark implementations are similar in being conventional ahead-of-time compilers that perform a series of compiler passes and optimisations. The passes are largely black-box optimisation passes, although SaC provides a variety of command line options to control the process, similar to conventional optimizing compilers. Both Futhark and SaC generate optimized C executables. The Futhark compiler is written in Haskell, while the SaC compiler is written in C. The main difference is that while SaC has an IO facility based on uniqueness types [94], and thus allows the construction of entirely freestanding SaC programs, Futhark does

Table 2.1: Table summarizing key differences between language implementations.

| | Accelerate | APL | Futhark | DaCe | SaC |
|---|---|---|---|---|---|
| **Implementation model** | | | | | |
| Standalone | ✗ | ✓ | ✓ | ✗ | ✓ |
| Implementation language | Haskell | APL | Haskell | C/Python | C |
| **Automatic Optimisations** | | | | | |
| Fusion | ✓ | ✓ | ✓ | ✓ | ✓ |
| Data layout | ✗ | ✗ | ✓ | ✗ | ✓ |
| Tiling | ✗ | ✗ | ✓ | ✓ | ✓ |
| AoS/SoA | ✓ | ✗ | ✓ | ✗ | ✓ |
| **Targets** | | | | | |
| CPU | ✓ | ✓ | ✓ | ✓ | ✓ |
| GPU | ✓ | ✓ | ✓ | ✓ | ✓ |
| FPGA | ✗ | ✗ | ✗ | ✓ | ✗ |
| Clusters | ✗ | ✓ | ✗ | ✓ | ✓ |

not have any IO facilities, and therefore no way to read input or write output. The output of the Futhark compiler is essentially a *library* that must be invoked by a program written in a general purpose language, through a C-based API. For convenience, the Futhark compiler provides a facility for automatically generating small wrapper C programs for benchmarking and testing purposes.

Accelerate is deeply embedded in Haskell as a Haskell library, and the compiler executes at runtime. Indeed, the Accelerate program is only constructed during the execution of the host Haskell program, where it is then compiled Just-In-Time (JIT) prior to execution. The Accelerate compiler automatically performs many optimisations and rewrites of the program, notably fusion. Although it is, strictly speaking, a JIT compiler, it is largely similar to traditional ahead-of-time compilers, and does not make use of tracing, deoptimisation, data-sensitive optimisation, or other dynamic behavior common in JIT compilers. It is, however, possible for the user to perform Haskell-level computation to specialize the Accelerate program for a given workload, which can be seen as a form of staged programming. This capability is exploited for some of the benchmarks we use, such as MG (Section 2.5).

DaCe is similar to Accelerate in being embedded in a frontend language (commonly Python), although it differs from the other language implementations in exposing far more control over the optimisation process, e.g., with the programmatic or visual tools outlined in Section 2.2.

APL is traditionally interpreted on CPUs, and the Dyalog interpreter provides efficient operations on arrays of different sizes. The Co-dfns APL GPU compiler

is written in data parallel APL to facilitate self-hosting on a GPU. It makes heavy use of runtime libraries to achieve performance, often calling specialized routines that implement APL primitives. Where most of the other compilers use sophisticated optimisations that may, however, produce unpredictable performance, the Co-dfns targets predictable performance. It provides tight integration with the Dyalog interpreter, allowing compiled modules to be used "in situ" from within the interpreter, while also producing standalone executables and shared objects that can be linked from other programming languages.

**Hardware targets** Finally, the language implementations all target multiple architectures, and all target multicores and CUDA/NVIDIA GPUs. The languages may also target other architectures, e.g., APL, DaCe and SaC also target clusters, and DaCe also targets FPGAs.

## 2.3 Benchmark Rationale, Evaluation Platforms, Methodology

### Benchmark Rationale

We illustrate functional array programming using four challenging benchmarks that represent a range of application domains and parallel computational models. Two benchmarks are from public suites, namely MG (numerical aerodynamic simulation) from NAS [16, 11] and Quickhull (computational geometry) from PBBS [7]. The other two are well-known algorithms, namely brute-force $N$-body (astrophysical simulation) and Flash Attention [60] (deep learning).

The benchmarks cover a *variety of code patterns, parallel constructs/structures, and optimisation techniques*: Quickhull exercises flattening of irregular nested parallelism, where manual application results in a composition of (all) flat-parallel skeletons—map, scan (prefix sum), reduce-by-index, scatter. The other benchmarks exhibit regular nested parallelism. MG is a 27-point stencil solved at different resolutions. Flash Attention has deeply nested structures of parallel and sequential loops that have matrix multiplication solvers at the innermost level.

The benchmarks exercise different optimisations. Quickhull's performance is determined by how well the flat-parallel skeletons are fused and mapped to the hardware. All other benchmarks require careful mapping of computation to different hardware levels, e.g., for achieving temporal reuse from fast memory and for enabling CPU vectorisation. Flash Attention requires tiling multiple dimensions into each level of the memory hierarchy. Finally, $N$-body exhibits an interesting case of dataset sensitivity on GPUs: the largest dataset benefits from sequentializing the inner function/loop, while the smaller datasets benefit from parallelizing all functions/loops.

Rather than choosing simple benchmarks that can readily be expressed, and effectively optimized in all five languages, we have selected a set of challenging benchmarks that often take a language outside its "comfort zone". For example, Quickhull utilizes scans (prefix sums), and neither SaC nor DaCe provide a scan

primitive. Several benchmarks require nested parallelism, and Accelerate does not normally provide it. MG uses stencils, and Futhark does not provide specific stencil support, nor deep learning support for Flash Attention. The hypothesis is that *collectively* these five research languages offer state-of-the-art solutions to both the expression and performance challenges posed by the benchmarks.

## Evaluation Platform: the Radboud Server

The evaluation is conducted on an experimental server at Radboud University that provides both a multicore CPU and a GPU. The server is booked for exclusive use during the experiments. The CPUs are two 16-core AMD EPYC 7313 CPUs running at 3.7 GHz. The peak performance of this machine is 1.895 Tflops (double precision). The RAM has a theoretical peak bandwidth of 204.8 GB/s, though experimentally (STREAM) we observe 140 GB/s.

The server also provides an NVIDIA A30 GPU, with peak performance of 10.3 Tflops and 5.2 Tflops for single and double precision, respectively, and 933 GB/s main memory bandwidth. We use NVCC version 12.1.66 and GCC 11.4.0.

For multicores, we use numactl ––interleave all to set the page-mapping policy and we configure the thread-binding policy of OpenMP to use OMP_PLACES="cores" OMP_PROC_BIND=true.

## Development Methodology

The benchmarks are implemented in each of the languages following an iterative procedure: first, we generate a correct implementation, and then repeatedly refactor it to optimize for the target hardware architectures. The final versions of the benchmark code are provided in a public repository[1] and represents the culmination of this process. The optimizing iterations are not described in detail, although we discuss the key optimisations for each language in the Sections 2.4 to 2.7.

In contrast to the separate CUDA and OpenMP baselines for each baseline *there is a single benchmark implementation in each language.* This implementation is compiled and optimized for both the multicore and GPU hardware, and for all datasets. The only exception is DaCe, which uses separate CPU and GPU implementations of $N$-body and Flash Attention. We discuss the development effort implications of having a single implementation in 2.8.

## Measurement Methodology

We measure total application runtime, but in the case of GPU execution, we exclude the time taken for the (1) CUDA-context initialisation, (2) CUDA-kernels compilation, and (3) host-device transfers of the program input and final result (only). For each benchmark and hardware platform, we perform five warm-up runs and then report the average of at least ten runs, or as many as required to reduce standard deviation to less than 3 %.

[1] https://github.com/diku-dk/CFAL-bench/

## 2.4 Benchmark 1: N-body Simulation

### Benchmark Description

The $N$-body simulation benchmark simulates how gravity acts on a system of $N$ bodies in $\mathbb{R}^3$. Each timestep computes the acceleration caused by gravity, and updates the bodies. Recall that each body is associated with multiple values, namely its position and velocity—each represented as a tuple of three double-precision floats—together with its acceleration and mass (both doubles). Hence, there are various ways to represent the bodies in memory, such as using AoS, SoA, or some combination.

### Baselines, Performance Measure, Datasets

The GPU baseline consists of two code versions, both using AoS layout, whose selection is optimally hardcoded for the target datasets. When the number of bodies $n$ is small ($n \leq 10^4$) we implemented a version that efficiently exploits both levels of parallelism by computing the acceleration for a body with a CUDA block of threads. For larger values of $n$ we use the CUDA-sample implementation [186] that uses only the outer level of parallelism. The key optimisation is 1D tiling in shared memory. That is, computing the acceleration for a body accesses all other bodies in the same order, so the computation is structured so that a chunk of bodies is collectively copied from global to shared memory by the threads in the same block and then reused. Without tiling, reuse is serviced by the L2 cache, which has higher latency and smaller throughput than GPU shared memory.

We have written a C with OpenMP CPU baseline that (i) parallelizes the computation of the `accel` function across processors using OpenMP and static scheduling, and (ii) ensures that the underlying (GCC) compiler successfully vectorizes the code by exploiting a SoA representation consisting of eight arrays of doubles.

We compute the number of flops for $n$ bodies and $t$ time steps as $(19n^2+12n)\cdot t$. Each interaction between two bodies is computed as

$$\frac{m_j(\vec{p_j}-\vec{p_i})}{(\|\vec{p_j}-\vec{p_i}\|^2+\epsilon^2)^{3/2}} \tag{2.1}$$

requiring 16 flops to compute the change in acceleration and another 3 additions to update it. The evaluation uses three datasets corresponding to the following values of $(n,t)$: $(10^3,10^5)$, $(10^4,10^3)$ and $(10^5,10^1)$. Note that these datasets all involve the same amount of work, but vary in how much parallelism is available.

### $N$-body Simulation with Futhark

The Futhark $N$-body simulation is expressed as two nested loops—the outermost a map and the innermost a reduce (fused with a map). Futhark's CPU and GPU backends both generate two code versions: (i) where the parallelism of the reduce is exploited (resulting in a segmented reduction), and (ii) where the innermost loop

is sequentialised. A code version is dynamically selected at run-time, as explained in 2.2. For the $N = 10^6$ case, the outer loop has sufficient parallelism and version (ii) is used.

Futhark's GPU backend tiles the sequentialized inner loop, as in the baseline implementation. This optimisation is not implemented for Futhark's CPU backend, which instead depends solely on the hardware caches. The source code uses an AoS layout, but the compiler always transforms it to the SoA layout.

### $N$-body Simulation with Accelerate

The acceleration between each pair of bodies is calculated by expanding the vector of bodies once along the rows, once along the columns (using replicate), and zipping the two resulting matrices with the accel function. All rows of the resulting matrix are then summed in parallel using a (rank-polymorphic) fold, resulting in a vector. Fusion is essential here as it would be inefficient, both in terms of runtime and memory allocation, to create the intermediate matrices. The fusion pass in the compiler ensures that the resulting code is a single nested parallel loop producing the output vector, without generating any intermediate arrays.

### $N$-body Simulation with Single Assignment C

To support vectorisation on the CPU, the SaC $N$-body code uses a struct for the $x, y, z$-coordinates instead of a double[3] array. Thus, the compiler eliminates the struct, and the generated code deals with three independent vectors of scalar coordinates. This ensures that the $x$-coordinates of consecutive bodies are contiguous in memory, and likewise for the $y, z$ coordinates and masses.

### $N$-body Simulation with APL

The points and velocities are each stored in an SoA model of nested arrays of 3 elements corresponding to the $x$, $y$, and $z$ coordinates. If the conceptual computation is thought of as a summation of each row of an $N \times N$ matrix, the APL code tiles the matrix into a series of $N \times B$ columns to compute a partial summation of each row a tile at a time. Each tiling pass first computes the differences $d$ for each dimension and then computes the weight matrix $w$ from these differences. To compute the velocity shifts based on these weights and each point's mass $m$, we utilize the expression APL(d×w)+.×m[ ], which leverages the APL+.× matrix multiplication in APL.

This particular ordering allows for each tile to first be computed as a fused kernel(s) of the differences and the weight computation, before the fast matrix-multiplication libraries are used in the final computations. This allows the code to make use of the facilities in the APL runtime, since fusion in Co-dfns only occurs on primitives and not on user-defined functions. It also enables portions of the problem to be mapped to fast matrix multiplication code in the runtime.

## $N$-body Simulation with DaCe

For CPU execution, we start with a $N$-body Python program and optimize the resulting SDFG with the custom workflow summarized in 2.2. The data are matrices of size $3 \times N$ and $4 \times N$, corresponding to a SoA representation.

For GPU execution, we use a Python program with two main differences compared with the CPU implementation. The first is additional tiling of one of the inner loops and the use of GPU-shared memory to reduce the number of loads from GPU-global memory. The second is the transposition of the matrices to produce an AoS representation and allow the compiler to vectorize the loads to GPU-shared memory.

## Summary

Table 2.2 summarizes the $N$-body compute rate for the five languages on both multicore CPU and GPU.

Table 2.2: $N$-body compute rate in Gflops (higher is better) for different numbers of bodies and iterations. The $n = 10^3$ case is done with 16 instead of 32 cores if that gave better results.

| | $n = 10^3$, $t = 10^5$ | | | $n = 10^4$, $t = 10^3$ | | | $n = 10^5$, $t = 10^1$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | CPU | | GPU | CPU | | GPU | CPU | | GPU |
| | 1C | 32C* | | 1C | 32C | | 1C | 32C | |
| Baseline | 19 | 280 | 560 | 19 | 580 | 720 | 19 | 610 | 1300 |
| Accelerate | 9.3 | 7.0 | 27 | 9.7 | 261 | 305 | 9.7 | 500 | 300 |
| APL | 1.5 | — | 13 | 1.7 | — | 14.0 | 1.3 | — | 15 |
| DaCe | 19 | 209 | 120 | 19 | 560 | 970 | 19 | 600 | 1600 |
| Futhark | 18 | 127 | 550 | 18 | 310 | 970 | 18 | 520 | 1600 |
| SaC | 19 | 240 | 39 | 19 | 560 | 230 | 19 | 600 | 260 |

**GPU performance** Futhark and DaCe beat the GPU baseline on the second and third datasets with speedup factors between $1.18 - 1.35\times$. Futhark is competitive with the baseline on the first dataset, but DaCe lags behind because it utilizes only the parallelism of the outer loop, which is insufficient. Accelerate and SaC come next, achieving fractions 5 %, 42 % and 23 % and 7 %, 32 % and 20 % of the baseline performance on the three datasets, respectively; the first dataset similarly suffers due to insufficient utilisation of parallelism. APL provides useful acceleration—in that its GPU compute rate is significantly higher than the sequential compute rate—but its performance is only as high as 3.5 % of the GPU baselines.

**Multicore performance** SaC and DaCe are competitive with the baseline on the second and third datasets (exceeding 95 %), and close on the first dataset (85 % and 75 %). Futhark and Accelerate provide good performance on the third (large)

dataset (exceeding 83 %) but perform less well on the second dataset (54 % and 45 %), and even worse on the first (45 % and 2.5 %).

DaCe presents the best blend of performance across the two hardware platforms. SaC is best developed for multicore execution, and its suboptimal GPU performance is directly related to its GPU backend receiving much less attention, and conversely for Futhark and Accelerate. These performance trends are reflected in the other benchmarks.

**Expression** The baseline consists of three low-level C++ code versions—two aimed at GPU and one at multicore execution—that use *different layouts* (SoA *vs.* AoS) and *manually-applied optimisations* (see Section 2.4). In comparison, the functional array languages use a *single* high-level code that is *automatically optimized* for multicore and GPU execution. While "beauty is in the eye of the beholder", we consider that all languages provide significantly more elegant code than the baseline—with a shoutout to APL for brevity—and encourage readers to make their own mind by examining our code repository.

## 2.5 Benchmark 2: MultiGrid (NAS MG)

### Benchmark Description

Multigrid methods solve differential equations numerically. The NAS parallel benchmarks [16] provide a simplified version of the V-cycle multigrid method to solve the discrete Poisson problem

$$\nabla^2 u = v \tag{2.2}$$

on a $n \times n \times n$ grid with periodic boundary conditions. The Laplacian $\nabla^2$ is approximated with a trilinear finite element discretisation $A$, which is defined as a function taking and returning an $n \times n \times n$ grid of real numbers:

$$\begin{aligned} A(x))[i_1, i_2, i_3] = \\ \sum_{j_1,j_2,j_3=0}^{2} w[j_1, j_2, j_3] \cdot x[(i_1 + j_1 - 1) \bmod n, \\ (i_2 + j_2 - 1) \bmod n, \\ (i_3 + j_3 - 1) \bmod n] \end{aligned} \tag{2.3}$$

for some weights $w \in \mathbb{R}^{3\times3\times3}$. This type of computational pattern is called a 27-*point stencil with periodic boundary conditions*.

We define a function f2c for projecting a fine grid on a coarser grid. This maps $x$ to $x(1 : n : 2, 1 : n : 2, 1 : n : 2)$, where we use the Fortran notation for the index range from 1 to $n$ with step 2, in each dimension. Similarly, we have a function c2f, which prolongates a coarse grid to a fine grid. This embeds some array $x$ of

**Algorithm 1** The V-cycle function $M$

```
1:  procedure M(r)
2:      if n ≠ 2 then
3:          rs = P(f2c(r))
4:          zs = M(rs)
5:          z = Q(c2f(zs))
6:          r = r − A(z)
7:          z = z + S(r)
8:      else
9:          z = S(r)
10:     return z
```

**Algorithm 2** MG

**Input:** $v$ : an $n \times n \times n$ array.
**Input:** $t$ : a number of iterations.
**Output:** $u$ : an $n \times n \times n$ array approximately satisfying $\nabla^2 u = v$.

$u = 0$
**for** $i = 1$ **to** $t$ **do**
  $r = v - A(u)$
  $u = u + M(r)$

Figure 2.3: Multigrid method for numerically solving $\nabla^2 u = v$.

size $n \times n \times n$ into an array $y$ of size $2n \times 2n \times 2n$ as $y(1 : n : 2, 1 : n : 2, 1 : n : 2)$. The remainder of $y$ are zeroes.

The computational core of MG is the V-cycle shown in Algorithm 1. This is a function composed of $A$ and three more 27-point stencils $P, Q, S$. The recursive call causes the stencils to operate on progressively coarser grids, until the input has shape $2 \times 2 \times 2$, after which the grid gets finer again. The MG benchmark, Algorithm 2, repeatedly uses the V-cycle function $M$ to update a solution to Eq. (2.2).

## Baselines, Performance Measure, Datasets

The CPU baseline selected is the Fortran + OpenMP implementation from NAS [16]. The GPU baseline selected is the CUDA implementation of NPB-GPU [11]. Both baselines perform an optimisation that exploits a domain-specific property of the $3 \times 3 \times 3$ array of weights, namely that the weights situated at the same distance from the center (at index $[1, 1, 1]$) are equal, i.e., the 27 weights contain only 4 distinct values. It follows that the number of flops per element can be significantly reduced from the 26 additions and 27 multiplications of a naive implementation, e.g., by performing rewrites of the form $w_1 a + w_1 b + w_1 c + w_1 d = w_1(a + b + c + d)$. Further, the number of additions can also be reduced by observing that for a portion of a slice in the $xy$-plane:

$$\begin{pmatrix} * & - & * \\ - & \circ & - \\ * & - & * \end{pmatrix}$$

the sum of the $*$s and $-$s are used to update not only $\circ$, but also the points directly above and below $\circ$. This can be exploited by precomputing, for each $(i, j)$, two temporary buffers containing the sums $x[i + 1, j, .] + x[i - 1, j, .] + x[i, j - 1, .] + x[i, j+1, .]$, and $x[i+1, j+1, .]+x[i-1, j-1, .]+x[i+1, j-1, .]+x[i-1, j+1, .]$. The

temporary buffers are then used to update $x[i, j, .]$. This technique not only reduces the number of operations, but also enables temporal reuse, e.g., the CUDA baseline allocates the temporary buffers in shared memory, thus reducing the number of accesses to global memory.

We report performance in $Gflops$ following the same methodology as in the NAS benchmark, where the number of flops for an $n \times n \times n$ grid and $t$ iterations is considered to be $58tn^3$. The evaluation uses the $ClassA$, $ClassB$, and $ClassC$ standard NAS datasets.

## MultiGrid with Futhark

The Futhark implementation supports both the naive and optimized versions of MG introduced in Section 2.5. A simplified code excerpt from the latter is shown in Figs. 2.4 and 2.5. In contrast to the lengthy baseline implementation (4662 source lines of code in Table 2.1) that consists of a multitude of distinct (specialized) computational kernels, the Futhark implementation abstracts (most of) the core computation of the 27-point stencil into *one* function that is suitably parameterized, e.g., over (i) the size of the result-array dimension $n$, (ii) an index-function representation $elmAt$ of the source array and (iii) the array of weights $ws$. The instantiations required to compute $Q(\texttt{c2f}(zs))$, $r - A(z)$ and $z + S(r)$ in Algorithm 1 are shown in Fig. 2.5, as functions named simply $Q$, $A$ and $S$, respectively. Finally, the function parameter $relaxKer$ may be instantiated with either the naive or optimized version ($relaxNAS$). On the cons side, Futhark does not support arbitrary recursion leading to a tedious (and ugly) implementation of procedure $M$ (of Algorithm 1) where recursion is modeled by a while loop that explicitly maintains the stack by means of updates to a jagged array.

The high-level implementation critically relies on the compiler to specialize the code. For example, Futhark's simplification engine aggressively inlines functions such as $relaxNAS$, $getEachElm$, $get8thElm$, $f$, $g$ at call sites, and performs a battery of transformations, such as copy propagation, constant folding, common-subexpression and dead-variable elimination, and scalar specialisation of array-literal indexing. For example, the specialisation of $relaxNas$ at the call site of $A$ (in Fig. 2.5), will eliminate the whole term starting with $ws[1] * (\ldots)$ appearing in the computation of $g$ (at lines 21-23)—because $ws[1]$ corresponds to constant 0 in this context, and multiplication with zero always results in zero.

Furthermore, as presented in Section 2.2, for the GPU case, the incremental flattening transformation will generate an "intrablock" code version that essentially maps the (inner) parallelism available within *computeRow* to the CUDA block level and the outer parallelism to the CUDA grid level, that is, the two nested *map*s at line 28. Consequently, the arrays $u1s, u2s$ produced by $tab\ n\ f$ (line 18) are placed in shared memory and reused from there in the computation of $tab\ n\ g$ (line 27). Finally, memory-related optimisations eliminate some of the array copying operations resulting from explicit modeling of the (recursion) stack.

## MultiGrid with Accelerate

The Accelerate implementation uses recursion to implement procedure M, despite Accelerate, like Futhark, not supporting recursion. It does so by exploiting the recursion provided by the Haskell host language via meta-programming. This unfolds the recursion, similar to how loop unrolling converts a loop to a sequence of instructions.

Accelerate features stencil primitives that simplify the implementation of relaxations. Again, using meta-programming, the weights are inlined into the stencil kernel.

Accelerate's fusion algorithm fuses f2c and c2f into a following stencil operation. This improves performance, while allowing the user to write the algorithm at a higher level.

## MultiGrid with Single Assignment C

SaC implements MultiGrid recursively, faithful to the specification. We use the generic stencil implementation shown in Fig. 2.6. This function generalizes to arbitrary dimension d and weights of arbitrary shape wshp. The modulo computations (mod) are optimized away by generating multiple partitions.

SaC's constant folder currently reduces the number of multiplications automatically, but not the number of additions. The Fortran version has one temporary array per thread, which it reuses. As SaC does not have explicit parallelism, getting this to happen is not straightforward. An alternative would be to compute the temporary arrays for all $(i, j)$ in parallel, but that is not worth it for cache reasons. The GPU code generation suffers greatly from a problem in the analysis that prevents arrays from being moved from and to the device.

## MultiGrid with APL

The APL implementation is a fairly direct translation of the pseudo-code:

```
M←{ 1:S st    z+S st  -A st z←Q st up( -1) dn P st  }
L2←{((+ , × )÷×  )*.5}
MG←{ L2  -A st  { +k M  -A st  } IT }
```

Where $MG$ is run for $IT$ iterations and the functions $st$, $dn$, and $up$ correspond to the stencil computation, downsampling, and upsampling, respectively. Downsampling and upsampling are implemented as direct array indexing operations:

```
dn←{ [i;i;i←1+2× ( )÷2]}                  Downsample
up←{X X[i;i;i←1+2×  ]← X←1e¯100 2× }   Upsample
```

APL does have a built-in stencil operator, but it is not suitable for high-performance optimisations such as those available in MG. We instead implement a custom stencil function $st$ that minimizes the number of arithmetic operations as described in Section 2.5. Since the entire function can be expressed as a single composition of arithmetic primitives and rotations, the resulting code can be fused into a single conceptual kernel:

```
  MG Stencil Operation
st←{k←
    x← ×k[0]
    x+←k[1]×(1 )+(¯1 )+r0←(1 )+(¯1 )+r01←(1 [1] )+¯1 [1]
    x+←k[2]×(1 r0)+(¯1 r0)+r1←(1 r01)+¯1 r01
    x+←k[3]×(1 r1)+¯1 r1
     x
}
```

## MultiGrid with DaCe

Since the SDFG IR does not support recursion, DaCe's MG implementation is based on a Python version [65]. We automatically optimize the CPU and GPU implementations, which fuses parallel computations (Map scopes) and tiles those that contain write-conflict resolutions to minimize atomic operations. However, as the auto-optimizer transforms the graph in a specific order, some optimisation opportunities may be missed. For example, Fig. 2.7 shows the MG's norm2u3 method and its DaCe-Python implementation, which includes two Map-Reduce patterns for computing s and rnmu. The auto-optimizer successfully fuses these computations into a single Map scope but cannot subsequently tile it because the relevant DaCe optimisation does not support Map scopes with write (WCR) conflicts to multiple data containers. To work around this limitation, we design a custom workflow. First, we apply the MapReduceFusion transformation repeatedly until no subgraph matches the optimisation pattern, resulting in two fused Map scopes, one for s and one for rnmu. Only then do we apply WCR tiling, which matches since each scope writes to a single data container.

## Summary

The NAS MG performance results for the five languages on both multicore CPU and GPU are summarized in Table 2.3. We follow the baseline performance measurement and report compute rate (Gflops).

Table 2.3: NAS MG compute rate in Gflops (higher is better) for NAS classes A, B, C.

| | Class A | | | Class B | | | Class C | | |
|---|---|---|---|---|---|---|---|---|---|
| | CPU | | GPU | CPU | | GPU | CPU | | GPU |
| | 1C | 32C | | 1C | 32C | | 1C | 32C | |
| Baseline | 8.7 | 83 | 190 | 9.0 | 95 | 240 | 8.2 | 48 | 240 |
| Accelerate | 1.6 | 7.0 | 72 | 1.70 | 8 | 71 | — | 14 | 82 |
| APL | 0.50 | — | 7.0 | 0.50 | — | 7 | 0.50 | — | 7.0 |
| DaCe | 3.3 | 28 | 130 | 3.6 | 41 | 140 | 3.00 | 37 | 230 |
| Futhark | 1.2 | 16 | 240 | 1.4 | 22 | 240 | 1.40 | 21 | 240 |
| SaC | 5.9 | 43 | — | 6.6 | 53 | — | 5.0 | 39 | — |

**GPU performance** Futhark offers constant performance across datasets (240 Gflops), which is competitive with the baseline for the two larger `B/C` datasets (99 %) and outperforms the baseline on the smallest `A` dataset (127 %). DaCe also offers competitive performance on the largest `C` dataset (95 %), but is less competitive for the first two datasets (70 % and 59 %). Accelerate offers relatively constant performance across datasets (71 %–82 %) reaching 30 %–39 % of the baseline performance. APL reports useful acceleration in comparison with the sequential execution, but reaches only 3 %–4 % of the baseline performance. For SaC, compiler shortcomings prevent GPU execution.

**Multicore performance** The baseline outperforms all of the functional array languages, likely due to suboptimal thread-binding policies. SaC performs best, reaching 52 %, 56 % and 81 % of baseline performance on the `A,B,C` datasets, followed by DaCe with 34 %, 43 % and 77 %. At a considerable distance come Futhark (19 %, 23 % and 44 %) and Accelerate (8 %, 8 % and 29 %), whose multicore backends are less mature than the GPU backends. Notably, Futhark exhibits very poor sequential performance but excellent scalability, albeit the latter is likely a case of "scaling the overhead".

**Expression** The Accelerate, APL, Futhark, and SaC teams found it infeasible to develop their code by following the long, low-level, and complex baseline code (4662 source lines of code, Table 2.1). The baseline specializes and hand-optimizes one generic stencil computation for a multitude of contexts. Instead, our implementation teams based their implementations on either the NAS specification or the SaC code, which is short, clean, and arguably very close to the algorithmic formulation (Section 2.5). Moreover, Fig. 2.4 shows that the key baseline optimisations, providing temporal reuse and a reduced number of arithmetic operations, can also be expressed generically and concisely without compromising GPU performance, as evidenced by Futhark's performance. While code comparisons are subjective, we invite the reader to reach a view by browsing our code repository [2].

## 2.6 Benchmark 3: Quickhull

### Benchmark Description

The Quickhull benchmark computes the convex hull of a finite set of points in 2-dimensional space. Algorithm 3 shows the pseudocode of the Quickhull algorithm: given a set $S$ containing at least 2 two-dimensional points, the convex hull of $S$ is computed by first (i) identifying the leftmost and rightmost points of $S$, named $A$ and $B$, respectively, then (ii) by partitioning $S$ into subsets $S_1$ and $S_2$ corresponding to the points below and above the line $AB$, and finally (iii) by combining the convex hulls of $S_1$ and $S_2$. The last step is facilitated by the $findHull$ helper function presented in Algorithm 4. The divide-and-conquer $findHull$ function takes two points $P$ and $Q$ and a set of points $S$ that are either all above or all below

[2] https://github.com/diku-dk/CFAL-bench/

line $PQ$. $findHull$ computes the point $C$ of $S$ that is furthest away from $PQ$, and hence must belong to the convex hull, then it eliminates from $S$ the points inside triangle $\Delta CPQ$ as they are guaranteed not to belong to the convex hull. The resulting set is partitioned into the points that are on the right- and left-hand side of lines $CP$ and $CQ$, named $S_l$ and $S_r$, respectively. The convex hull of $S$ consists of point $C$ together with the convex hulls of $S_l$ and $S_r$.

Such divide-and-conquer recursion induces *irregular nested parallelism*: the recursive calls form a binary tree where each call performs parallel operations (e.g., filtering, partitioning) on arrays of different sizes. Importantly, all the nodes at the same level in the tree can be processed in parallel.

Ideally, the Quickhull implementation should resemble the divide-and-conquer structure of Algorithm 4. This approach was pioneered in the NESL language [31], and can be automatically mapped to hardware using flattening transformations [33], and may target multicore [47] or GPU architectures [247, 23]. Since none of the studied languages support flattening of irregular parallelism, their corresponding implementations are derived by manually flattening the code corresponding to Algorithm 4.

## Baselines, Performance Measure, Datasets

The CPU baseline selected is the convex-hull implementation of the PBBS benchmark[3] [7], that offered state-of-the-art multicore performance at the time of writing the paper this chapter is based on. The implementation uses a fork-join (task-based) strategy that dynamically decomposes and exploits both levels of parallelism while tasks have good granularity. This strategy results in cache-friendly behavior that cannot be replicated by any of the studied languages, given their data-parallel paradigm. This paradigm effectively enforces barriers around the parallel processing of the nodes at each level in the tree, resulting in larger re-use distances. We also measure VQhull from Chapter 4.

We do not use a GPU baseline as, other than the studied languages, we could not find a publicly available implementation competitive with the PBBS implementation. This emphasizes the importance of high-level data-parallel models offering efficient hardware mappings, e.g., to GPUs.

We follow PBBS in reporting performance as runtime. The evaluation uses three PBBS datasets, each consisting of a set of one hundred million points that are uniformly distributed in the interior of a *Rectangle*, in the interior of a *Circle* (a disk), and on a *Quadratic* curve. Their convex hulls consist of 49, 1681, and 548136 points, respectively.[4]

## Quickhull with Futhark

The Futhark implementation of Quickhull has been manually flattened to be expressible with flat data parallelism. At a high level, it consists of an outer while loop that tracks the points determined to be part of the hull, as well as a sequence

[3] https://github.com/cmuparlay/pbbsbench

[4] The datasets are named as in PBBS and are generated with the PBBS infrastructure.

of sets of points that have not yet been classified. This sequence is conceptually a multidimensional jagged array, and is represented using a standard flag array approach. The implementation consists of a composition of map, reduce-by-index, filter, and partition operations, where the implementation of the latter two relies on scan (a.k.a., prefix sum) and scatter (a.k.a., parallel write). Efficient execution on GPU hardware is enabled by specialized code generation for scan and reduce-by-index (Section 2.2).

## Quickhull with Accelerate

The Accelerate implementation, like Futharks', is also flattened manually, making it necessary to maintain some segment information. The main work during one step is a parallel filter operation, applied to all segments. This consists of a mapping operation to determine which points to discard, a segmented scan over the resulting flag vector to compute the target indices, and a subsequent permutation.

## Quickhull with APL

The APL version first reorders the points so that membership in the hull set can be tracked using a Boolean mask to partition the data vectors into segments. On each iteration, this mask is used to group each point and determine membership within the hull set. Points falling within the hull are removed, while the new point that divides each segment is computed and added to the set by flipping its bit in the bitmask.

APL currently lacks a method for computing Scans or Reductions over sub-segments of an array indicated by a key. Instead, the expression APLM[R K[M← D]] computes the first indices of $D$ that have the maximum $D$ for each unique key in $K$. This expression is reasonably efficient, but involves sorting two vectors to find the max, which is significantly less efficient than a max reduction might be. Moreover, the above code does not split out the hull elements from elements still to be determined; as a result, if the number of points on the hull is large, we may redundantly compute distances that do not matter over time, leading to reduced performance when the hull contains many points. Finally, reordering the points at the beginning instead of at the end requires sorting a much larger array.

## Quickhull with SaC

We implement the algorithm recursively. As SaC's syntax is very close to C, we can straightforwardly adapt the partition scheme from Edelkamp [71]. We also find the furthest points in the same loop that partitions $S$ into $S_l$ and $S_r$ to reduce the number of passes over the data. As PBBS performs a Hoare partition and does not merge these loops, the SaC implementation is at least twice as fast sequentially. PBBS also works with indices instead of permuting the array, which gives bad cache behaviour for deep recursions. For this reason SaC is even five times faster for the Quadratic data set. The compiler is not able to parallelize

recursive calls. We are not able to implement the flattened algorithm efficiently as SaC lacks parallel writes. This means we have no parallel results.

## QuickHull with DaCe

DaCe does not support recursion and, therefore, cannot directly implement Algorithm 4. The alternative formulation using flattening transformations, described in Section 2.6, could be implemented in DaCe by composing control flow (for-loops) with map scopes and reductions to compensate for the lack of (segmented) scan primitives. However, such an implementation would be tedious and uncompetitive with languages that can express those parallel paradigms naturally. The Library Node mechanic could also be leveraged to map a high-level (segmented) scan representation directly to optimized calls, e.g., to the NVIDIA Thrust library for GPUs. Overall we consider the additions required to facilitate a reasonable QuickHull implementation in DaCe to be beyond the scope of this work.

## Summary

It is not feasible to develop parallel Quickhull implementations in DaCe or SaC, mainly because they lack support for a key operation like prefix-sum or scatter, or for fork-join parallelism. The Quickhull performance results of Accelerate, APL, and Futhark on multicore CPU and GPU are summarised in Table 2.4. We follow the baseline performance measurement and report runtime (in seconds).

Table 2.4: Quickhull runtime in seconds (lower is better) for $10^8$ points sampled uniformly from three geometric shapes.

| | Circle | | | Rectangle | | | Quadratic | | |
|---|---|---|---|---|---|---|---|---|---|
| | CPU | | GPU | CPU | | GPU | CPU | | GPU |
| | 1C | 32C | | 1C | 32C | | 1C | 32C | |
| Baseline | 4.4 | 0.2 | — | 3.4 | 0.11 | — | 35.1 | 2.9 | — |
| VQhull | 0.71 | 0.1 | — | 0.55 | 0.08 | — | 4.4 | 0.32 | — |
| Accelerate | 7.4 | 1.6 | 0.16 | 3.6 | 1.2 | 0.11 | 48 | 13 | 4.3 |
| APL | 22 | — | 1.2 | 15 | — | 0.69 | 110 | — | 7.6 |
| DaCe | — | — | — | — | — | — | — | — | — |
| Futhark | 5.6 | 1.3 | 0.064 | 3.8 | 1.2 | 0.047 | 38 | 4.0 | 0.68 |
| SaC | 1.8 | — | — | 1.6 | — | — | 6.5 | — | — |

**GPU performance** Futhark outperforms the multicore baseline, achieving 313 %, 234 % and 430 % of its performance. Accelerate is competitive with the baseline on the Circle (125 %) and Rectangle (96 %) datasets, but falls behind on Quadratic (68 %). While APL reduces runtime compared with its sequential performance, it reaches only 16 %, 16 % and 39 % of the baseline performance on Circle, Rectangle, and Quadratic datasets.

**Multicore performance** The Accelerate and Futhark implementations are obtained by manually applying a flattening transformation [33] to the divide-and-conquer (irregular) parallelism exposed by Quickhull. The baseline outperforms all languages as it uses a fork-join style that dynamically spawns parallel tasks that have good granularity and temporal reuse (Section 2.6). Futhark achieves 16 %, 10 % and 72 % and Accelerate 13 %, 9 % and 23 % of the baseline performance on the three datasets, respectively.

**Expression** It seems that Quickhull is best expressed in a divide-and-conquer parallel style, similar to Algorithm 4, where the text at lines 3 and 5–8 is implemented with data-parallel constructs such as map-reduce and partition. This would offer a clean and algorithm-faithful high-level specification that preserves the opportunities for efficient mapping to different architectures—e.g., by specializing the flattening transformation for multicore [47] or GPU [247, 23] execution. In principle, this would produce (1) efficient multi-core code resembling the baseline implementation and (2) efficient GPU code resembling the one of Accelerate/-Futhark. We consider that our implementations are shorter, easier to understand and more high-level than the baseline. For example, they contain no notion of memory or memory management, and provide correct-by-construction parallelism enabled by implicitly-parallel constructs.

## 2.7 Benchmark 4: Flash Attention

### Benchmark Description

Self-Attention is a machine-learning mechanism for finding dependencies among inputs in a sequence [235]. We consider a sequence of $N$ words or tokens, each with an embedding $\mathbf{v}_i, i \in 1..N$ of length $M$. We want to find another embedding $\mathbf{o}_i$ for each word that encodes its relationship to the other tokens. To that end, we select one word, the query, and we compute its similarities $\mathbf{s}_{ij}, j \in 1..N$ to each other token as the dot product of their embeddings, $\mathbf{v}_i \cdot \mathbf{v}_j$. We normalize these similarities using Softmax [40], converting them to a probability distribution. Multiplying the result vector $\mathbf{p}_i$ with the initial word embeddings produces the final result.

Machine learning architectures, like transformers, learn weights for the different embedding uses: $\mathbf{Q}$ for query words, $\mathbf{K}$ for the similarity computation, and $\mathbf{V}$ for the output embedding. In Multi-head Attention, multiple heads compute the attention corresponding to distinct parts of the word embeddings in parallel, each with size $d$. Hence, in Algorithm 5, the weight matrices have size $N \times d$. Standard Self-Attention materializes all of the $\mathbf{S}$ and $\mathbf{P}$ matrices, which have size $N \times N$. In applications of interest, the sequence is large enough to consider $N \gg d$, and $Q$, $K$, $V$, and $O$ are all tall and skinny matrices. Multiplying such matrices exhibits low arithmetic intensity (computation is $O\left(N^2 d\right)$, but memory operations are $O\left(N^2\right)$), making self-attention memory-bound.

Softmax, shown in Algorithm 7, normalises the similarities matrix $\mathbf{S}$ by computing for every row $i$ two quantities: the maximum value $m_i$ and the normalisation term $l_i$, defined as $l_i = \sum_{j=1}^{N} \exp(s_{i,j} - m_i)$. The output probabilities $\mathbf{P}$ are computed as $p_{i,j} = \exp(s_{i,j} - m_i)/l_i$ to avoid overflow. Overall, each input row must be read three times since the dependency on $m_i$ disallows loop fusion. An alternative formulation called Online Softmax [175] and shown in Algorithm 8 removes this constraint by using exponent rules to compute $l_i$ recursively, reducing the required reads of the input and enabling computation of the output in blocks.

Flash Attention [60] utilizes Online Softmax to tile the matrix multiplications and the normalisation operation, fuse them together, and perform the overall computation in $T_i \times T_j$ blocks. In this reformulation, the intermediate results are small enough to fit in fast memory (e.g., registers and CUDA shared memory), dramatically reducing accesses to slower global memory. An implementation of the algorithm in DaCe data-centric Python is shown in Fig. 2.11.

Finally, Algorithm 6 presents a midpoint between Standard and Flash Attention, dubbed Custom Attention, that blocks the computation over $d \times d$ slices of $Q$ and computes corresponding slices of the result $\mathbf{O}$. Custom Attention does not serialize computation, but also does not guarantee that the intermediate matrices ($\mathbf{Sb}$ and $\mathbf{Pb}$ of size $d \times N$) can be maintained in fast memory.

## Baselines, Performance Measure, Datasets

The reference Flash Attention implementation [61] supports half-precision arithmetic (fp16 and bf16) and utilizes the specialized tensor (matrix) core units on NVIDIA (AMD) GPUs. Since not all languages support half-precision arithmetic or execution on tensor cores, we write our own GPU baseline using single-precision arithmetic and the regular FPUs. Following the high-level algorithm in Fig. 2.11, each $T_i \times d$ block of the output $\mathbf{O}$ is assigned to a single thread block. The computation is done in a single kernel in $T_j$ steps and utilizes GPU-shared memory and registers to reduce loads from GPU-global memory. Each thread block reads its assigned $\mathbf{O}$ and $\mathbf{Q}$ blocks and the whole $\mathbf{K}$ and $\mathbf{V}$ matrices. Therefore, $\mathbf{O}$ and $\mathbf{Q}$ are read only once, while $\mathbf{K}$ and $\mathbf{V}$ are loaded as many times as the number of thread blocks. All intermediate results, for example, the maximum values $m$ and normalisation factors $l$, are written to shared memory and registers. The implementation further 2-D tiles the computation of each thread block and utilizes memory coalescing to achieve high GPU-global memory throughput.

The CPU baseline tiles the computation in the same way. Each $T_i \times d$ block of the output $\mathbf{O}$ is assigned to a single CPU thread. The matrix multiplications are executed with a specialized single-threaded implementation optimized for the AMD Zen architecture.

We report the performance in *Gflops* and consider only the operations present in the standard Self-Attention algorithm when measuring the workload, i.e., $N^2(4d + 5)$ floating point operations. Our evaluation uses four randomly generated datasets with the following parameters: (1) $d = 64, N = 16,384$, (2) $d = 64, N = 32,768$, (3) $d = 128, N = 8,192$, and (4) $d = 128, N = 16,384$.

### Flash Attention with Futhark

Futhark's implementation in Fig. 2.10, closely resembles Custom Attention, Algorithm 6. During GPU compilation the incremental flattening transformation distributes the outer map processing each $d \times d$ block of $Q$ (line 13) across the computations of (i) the transposed matrix multiplication `mmmT` at line 10, (ii) the Online Softmax of Algorithm 8 at line 11, and (iii) the second matrix multiplication `mmm` at line 12. The resulting batch matrix multiplication kernels (i & iii) are optimized for spatial and temporal locality using block and register tiling. Online Softmax (ii) is similarly decomposed into an intragroup kernel that performs the computation of $m_i$ and $l_i$ in shared memory, and another kernel that updates the elements of $P$.

The CPU compilation diverges early from the GPU compilation and does not perform locality optimisations, resulting in very poor performance. Notably, the compilation of Custom Attention manifests the intermediate $P$ matrix in global memory, and hence cannot match the performance of the baseline that implements the more efficient Flash Attention algorithm.

### Flash Attention with Accelerate

Accelerate implements Custom Attention, Algorithm 6. We find that the performance is sensitive to the choice of block size. Matrix multiplication is implemented naively in the benchmarks. A faster alternative would be to use the matrix multiplication available through Accelerate's BLAS binding.

### Flash Attention with Single Assignment C

SaC implements Algorithm 6, but does not yet provide register tiling. The GPU backend is not able to generate a kernel for all computations. Moreover, the code does not vectorize on the CPU. That would require a non-scalar fold, which we cannot implement efficiently.

### Flash Attention with APL

The APL code is a fairly direct translation of the FlashAttention algorithm, but cannot control memory allocation and, therefore, cannot exploit the inner loops working on shared memory objects. Hence, the implementation is closer to Algorithm 6. Input values are converted to 3-dimensional arrays to facilitate blocking before entering the main loop. While the other languages can express single precision floating point computation, existing APL implementations only support double precision floating points or greater.

### Flash Attention with DaCe

For CPU, we use the Python code shown in Fig. 2.11 with auto-optimisation heuristics. DaCe generates OpenBLAS calls for the matrix multiplications in lines 16 and 25). We explicitly set the number of OpenBLAS threads to one, resulting

in a very similar implementation to the CPU baseline code. For GPU, we use the simpler implementation shown in Algorithm 6.

## Summary

The Flash Attention performance results for the five languages on both multicore CPU and GPU are summarised in Table 2.5. We follow the baseline performance measurement and report compute rate (Gflops).

Table 2.5: FlashAttention compute rate in Gflops (higher is better) for different $N$ and $d$.

| | $(d, N) = (64, 16384)$ | | | $(d, N) = (64, 32768)$ | | |
|---|---|---|---|---|---|---|
| | CPU | | GPU | CPU | | GPU |
| | 1C | 32C | | 1C | 32C | |
| Baseline | 78 | 2000 | 5700 | 77 | 2200 | 6600 |
| Accelerate | 73 | 96 | 240 | 150 | 400 | 570 |
| APL | 15 | — | 270 | 15 | — | 300 |
| DaCe | 72 | 1700 | 2100 | 73 | 1900 | 1300 |
| Futhark | 4.0 | 84 | 3700 | 4.0 | 84 | 3700 |
| SaC | 26 | 670 | - | 26 | 710 | - |

| | $(d, N) = (128, 8192)$ | | | $(d, N) = (128, 16384)$ | | |
|---|---|---|---|---|---|---|
| | CPU | | GPU | CPU | | GPU |
| | 1C | 32C | | 1C | 32C | |
| Baseline | 81 | 2200 | 5400 | 80 | 2400 | 6600 |
| Accelerate | 18 | 48 | 110 | 36 | 180 | 230 |
| APL | 25 | — | 360 | 25 | — | 570 |
| DaCe | 88 | 1600 | 3000 | 88 | 2100 | 3700 |
| Futhark | 3.0 | 73 | 4600 | 2.0 | 67 | 4400 |
| SaC | 27 | 610 | - | 26 | 700 | - |

**GPU performance** All languages implement the less efficient Custom Attention (Algorithm 6), and hence cannot match the baseline performance. Futhark and DaCe are the closest, reaching (65 %, 56 %, 85 % and 67 %) and 37 %, 20 %, 55 % and 56 % of the baseline performance on the four datasets respectively. APL and Accelerate lack locality optimisations, which restricts the percentage of baseline performance to (4.7 %, 4.5 %, 6.7 % and 8.7 %) and (4.1 %, 8.7 %, 2.5 % and 3.5 %), respectively. Compiler shortcomings prevent a SaC implementation.

**Multicore performance** DaCe reaches a good fraction of the baseline performance 82 %, 85 %, 73 % and 90 %, followed by SaC at 33 %, 32 %, 22 % and 30 %. Accelerate and Futhark suffer from a lack of locality optimisations, reaching only

(4.7 %, 18 %, 2.2 % and 7.7 %) and (4.1 %, 3.8 %, 3.3 % and 2.8 %) of the baseline performance. We did not benchmark APL using manual task parallelism for multicore CPUs, and the implicit multi-threading of the interpreter did not significantly impact results, so we include only a single result for each CPU result for APL.

**Expression** While Custom Attention is elegantly implemented by all languages, with code that closely matches Algorithm 6, we cannot meaningfully compare most languages with the baseline because it implements the more efficient Flash Attention. The exception is DaCe's Multicore implementation of Flash Attention (Fig. 2.11) that is very close to the algorithmic specification.

## 2.8 Benchmarking Summary and Analysis

Measuring the performance of the five languages on such a challenging set of benchmarks on both a multicore and a GPU exposes their design and implementation strengths and weaknesses. Broadly speaking, a language delivers good performance on a target architecture if it has the right constructs for specifying the benchmark, paired with the right optimisations to generate efficient code for the target architecture. Conversely, performance is lacking if the language lacks the required constructs, optimisations, or, indeed, an implementation for the target architecture.

We find that *functional array languages are expressive: they can represent a variety of array applications both concisely, and preserving a close relationship to the high-level specification.* Detailed evidence for this claim is provided in the summary of each benchmark, i.e., in Sections 2.4 to 2.7, and some specific examples follow. SaC's MG implementation closely follows the mathematical formulation presented in Section 2.5 and Algorithm 1. Notably, it defines one two-line generic stencil kernel (Fig. 2.6) that is instantiated in different contexts (i.e., to perform the computations denoted by $A$, $P$, $Q$, $S$). In contrast, the CUDA/FORTRAN baselines specialize each stencil kernel into different code ultimately resulting in a code base that our benchmark implementation teams found extremely hard to understand. Futhark builds on SaC's implementation and demonstrates that the baseline's optimisations can be expressed at a high level (i.e., applied to the generic stencil kernel), as evidenced by the competitive GPU performance. DaCe's base $N$-body implementation consists of about ten lines of Python code, similar to the five equations describing the problem. In contrast, the baseline consists of three code versions (one for CPU and two for GPUs) that are optimized in very different ways, e.g., they utilize different levels of parallelism and different array layouts (SoA *vs.* AoS). For Quickhull, Accelerate and Futhark present reasonably straightforward manually-flattened implementations that build on their rich set of second-order array operators and guarantee deterministic-by-construction parallelism. Ultimately, all arguments related to "elegant expression" are subjective, and we encourage readers to inspect our code repository and draw their own conclusions.

| | Accelerate | APL | DaCe (CPU) | DaCe (GPU) | Futhark | SaC | Baseline (CPU) | Baseline (GPU) |
|---|---|---|---|---|---|---|---|---|
| *N*-body | 113 | 25 | 76 | 53 | 46 | 61 | 164 | 411 |
| MG | 137 | 27 | 255 | | 151 | 136 | — | 4662 |
| Quickhull | 254 | 26 | — | — | 161 | 203 | 208 | — |
| FlashAttention | 179 | 21 | 31 | 26 | 90 | 79 | 1280 | 908 |
| Total Codebase | 683 | 99 | — | — | 448 | 479 | 1652 | 5981 |
| | | | | | | | 7633 | |

Table 2.6: Benchmark sizes in Source Lines of Code (SLOC).

As an approximate measure of code size and development effort we use Source Lines Of Code (SLOC), a crude but widely accepted measure [204]. SLOC counts the lines of code omitting blank and comment lines. Table 2.1 compares the benchmark sizes. The precise numbers must be taken with a grain of salt, as they do not account for differences in personal style and tooling—for example, some languages use external tooling for timing and input processing, while others implement it for each benchmark. The major trends, however, are not obscured by such issues. A major difference is that the functional array programmer writes a single program that is compiled for multiple targets, where there are separate CPU and GPU baselines. Hence the bottom two rows of the table show that *at 7633 SLOC the total baseline codebase is much larger, at least 10× than any of the functional array codebases (683 SLOC).* The functional code requires fewer lines of code than the baselines in every case except for Accelerate Quickhull on CPU. *The functional codebases are at least* $2\times$ *smaller than the CPU baseline codebase (683 vs 1652 SLOC) and much smaller, at least* $8\times$*, than the GPU baseline codebase (683 vs 5981 SLOC).* Of the functional languages APL is the most concise (99 SLOC in total), while Accelerate is the most verbose (683 SLOC in total).

Regarding GPU and multicore performance, there are a total of 36 benchmark instances with a baseline. In 30 % of the instances (11 instances), at least one language matches or outperforms the baseline. In 70 % of the instances (25 instances), at least one language achieves more than 80 % of the baseline performance. In a further 25 % of the instances (9 instances), at least one language achieves more than 50 % of the baseline performance. In only 6 % of instances (2 instances), no language achieves more than 50 % of the baseline performance. These instances are Quickhull solving the Circle and Rectangle datasets on a 32-core CPU, and the reasons are elaborated in Section 2.6. *The key conclusion from these results is that they demonstrate that mature functional array languages have the potential to deliver performance competitive with the best available conventional techniques.*

The remainder of the section briefly summarizes and analyses the performance

of each language on the set of benchmarks and target architectures.

## Futhark Summary and Analysis

Futhark's GPU performance is competitive with the baselines for all benchmarks other than Flash Attention. That is, Futhark sometimes outperforms the baseline, and the speedup is always more than $0.97\times$ the baseline. A key factor for achieving good performance is the compiler's ability to exploit nested parallelism at the application level by mapping it to different levels of the hardware hierarchy. The key optimisation is *incremental flattening* ($\mathcal{IF}$), a multi-version compilation technique (Section 2.2). For $N$-body, $\mathcal{IF}$ generates two versions of the code, one that utilizes only the outermost level of parallelism for small numbers of bodies, and one that parallelizes both outer and inner levels of parallelism for large numbers of bodies. For MultiGrid andFlash Attention, $\mathcal{IF}$ generates code versions that map the innermost level of parallelism to the CUDA thread block level, enabling reuse from fast memory.[5] Finally, Quickhull benefits from efficient GPU implementations of flat-parallel constructs such as scan and reduce-by-index, and Flash Attention from block-and-register tiling optimisation of batched matrix multiplications. All benchmarks benefit from fusion [103], which is enabled by IR constructs that allow, e.g., to also return the mapped part of a map-reduce composition [102], if this is needed elsewhere.

Futhark's CPU backend uses a different compilation pipeline that has received less attention than the GPU. Notably, it lacks locality optimisations, resulting in abysmal performance for Flash Attention, and it employs a runtime system that aims to dynamically adjust the levels of parallelism that are mapped to the hardware, but which introduces significant overhead for $N$-body and MG.

## Accelerate Summary and Analysis

All the Accelerate benchmark programs are straightforward implementations of the algorithms, without any attempt to tune them for performance. Furthermore, none of the benchmarks utilise bindings to high-performance libraries like BLAS. Accelerate's foreign function interface makes it easy to use such high performance libraries. The results reveal that Accelerate has a relatively high performance overhead, although for large computations the overhead is amortized. The Accelerate compiler pipeline is currently being completely rewritten. As a consequence, the back-ends haven't received major updates for several years. In particular, the GPU back-end has not been optimized for more modern GPU architectures, and the effect can be seen in the benchmarks.

## SaC Summary and Analysis

SaC's CPU performance is competitive with the baseline in $N$-body. For both Multigrid and Flash Attention the source code of SaC is much closer to the math-

[5] As such versions may run out of resources (e.g., fast memory) for datasets with a large innermost parallel dimension, they require multi-version compilation.

ematical specification than the baseline. This clarity costs a factor 1.2 – 3.9 performance. The key optimisation is *with-loop folding*, which improves the temporal locality by composing functions element-wise.

The SaC GPU backend is not as well developed and the benchmark performance does not come close to the baseline.

## APL Summary and Analysis

APL is by far the most concise language, and the computational core of each benchmark is implemented using between 10–14 lines of code. The amount of native data type, functional, and structural flexibility in the APL language exceeds that of the other languages in this comparison. It also has the richest native set of primitive operations over its core data structure, the array.

However, both the Dyalog CPU interpreter and Co-dfns GPU compiler lack the full set of optimisations to extract the maximum performance from these benchmarks. In both implementations, the primary factor limiting performance is the inability to eliminate memory access costs associated with intermediate and temporary variables. For instance, the current runtime requires the use of double-precision floating points, which harms performance on the Flash Attention benchmark. Likewise, because the GPU runtime depends on runtime level JIT compilation provided by external libraries [165], there is some variability and lack of predictability in the level of fusion and the kind of laziness that will manifest. This causes some benchmarks to perform worse than strictly necessary for the given APL code. For the QuickHull benchmark, runtime JIT does not optimize the implementation of key-based reduction, which must be implemented using other primitives, since APL does not presently support key-wise reduction. This greatly increases both the number of kernels and the main memory throughput required. APL's dynamic typing makes type inference more difficult, limiting the ability to statically specialize the output to known data sizes. This requires runtime verification of some data types, array shapes, and data ranges to implement the primitives, further increasing overheads.

While it is possible to manually tile or tweak the APL code to encourage the runtime JIT to produce more optimal code, doing so is unidiomatic. The benchmarks are expressed as idiomatic APL without such manual optimisation.

The Co-dfns GPU compiler is relatively new, and is the least optimizing compiler among those evaluated here. There is very little optimisation of locality between kernels or between various computations. So when a computation maps to a specific library function it is performant, but combining unfused primitives typically requires writing to GPU main memory, limiting performance. Extending Co-dfns with additional optimisations would permit more efficient kernel generation and resolve many of these issues.

While the lack of optimisations limits APL performance, the Co-dfns compiler consistently delivers more than a $10\times$ performance improvement over the Dyalog CPU interpreter. This shows that APL is eminently suitable for GPU execution, and the ease with which it can be translated for the GPU, even without optimisations and specialized compilation pipelines. This is rare, since most languages

rely on specific compiler optimisations to achieve good GPU performance.

### DaCe Summary and Analysis

DaCe's greatest asset is undoubtedly its extensive optimisation arsenal. The automatic tools frequently lead to acceptable performance, while experimenting with different parallel schedules and data layouts is often relatively easy. At the same time experienced users can further improve performance by manually applying transformation workflows, by editing the graph representation with visual tools, or even by embedding scheduling decisions in the high-level code. These capabilities can deliver high performance, as shown in $N$-body, where DaCe matches the baseline on CPU and surpasses it on GPU by35 % and 23 % on the medium and large datasets, respectively. However, taking advantage of such tools requires both a level of expertise and investing some time.

Another of DaCe's strengths is its Library Node system, which simplifies integration with optimized libraries like vendor-provided BLAS/LAPACK implementations. This leads to high performance in benchmarks such as Flash Attention, where DaCe achieves 80 %–90 % of the baseline's performance on CPU.

A significant drawback for some algorithms, however, is the lack of support for recursion. For example the lack of recursion constrains DaCe to an imperative implementation of MG, and makes the generation of a graph representation a difficult task in Quickhull.

## 2.9 Discussion and Conclusion

Functional array languages offer the enticing prospect of the high-level specification of correct-by-construction parallel array programs that generate performant and portable code. Hence, they are attracting research interest, and are emerging as a class of high-performance parallel languages. This chapter compares the design, implementation, and performance of five functional array languages, namely Futhark, Accelerate, SaC, APL, and DaCe, and makes the following research contributions.

We make a systematic comparison of the designs of the five functional array languages. All of the languages support rectilinear arrays, several support rank polymorphism, and user-defined element types, but only APL supports jagged arrays and heterogeneous arrays. All our languages except APL use static types, most support parametric polymorphism, and only Futhark supports dependent types. Although most languages offer at least partial determinism, the languages otherwise offer a wide variety of parallel paradigms: parallelism may be implicit or explicit, may be nested, and task parallelism may be available alongside the data parallelism.

We outline the implementation of the five functional array languages, and make a systematic comparison. There are a variety of implementation techniques: the Futhark and SaC compilers, the Accelerate DSL, the DaCe framework, and the APL interpreter and compiler. All implementations target multicore CPUs and

GPUs, and several languages target other platforms. Although all implementations fuse computations where appropriate, implementations otherwise support different sets of optimisations.

We demonstrate the expressiveness of functional array programming using four challenging benchmarks that represent a range of application domains, parallel computational models, and exploit different optimisations (Sections 2.4 to 2.7). The benchmark code is developed iteratively by applying a sequence of optimisations to an initial correct implementation.

We compare the codebase sizes, and hence approximate the programming effort, of the benchmarks in the functional array languages and in the OpenMP and CUDA baselines. We use Source Lines Of Code (SLOC) as a crude but widely accepted measure [204] (Table 2.1). A major difference is that the functional array programmer writes a single program that is compiled for multiple targets, where there are separate CPU and GPU baselines. Hence *the total baseline codebase (7633 SLOC) is much larger, at least 10× than any of the functional array codebases (683 SLOC).* The functional code requires fewer lines of code than the baselines in all but one case. *The functional codebases are at least 2× smaller than the CPU baseline codebase and much smaller, at least 8×, than the GPU baseline codebase.* Of the functional languages APL is the most concise, with Accelerate and SaC being the most verbose.

One reason for their conciseness is that the functional array code omits architectural aspects like the memory hierarchy and how the algorithm is mapped to hardware. This enables the development of code without intimate knowledge of the parallel hardware. It also allows the implementation to generate both multicore and GPU programs from a single source. Thus, *functional array code should be more easily ported to, and optimized for, new parallel architectures.* The challenge of porting functional array programs to some new hardware, perhaps neuromorphic computing, is that of writing a performant implementation, like a compiler backend, for the new hardware. The evidence for this claim in Sections 2.4 to 2.7 is partial as only a few languages, notably Dace and Futhark, consistently achieve good performance on both multicore and GPU. We argue that the primary reason that other languages don't deliver good performance on both architectures is language engineering: the research teams have not yet had the resources to engineer a well performing implementation.

We summarize and analyze the multicore CPU and GPU performance of the languages on 39 instances of the four open-source benchmarks [143] to explore why each language performs well, or poorly, on each benchmark and architecture (Section 2.8). We find that in 30 % of the instances, at least one language matches or outperforms the baseline; that in 70 % of the instances at least one language achieves more than 80 % of the baseline performance; and in 94 % of the instances at least one language achieves more than 50 % of the baseline performance. In only 6 % of instances, no language achieves more than 50 % of the baseline performance. These results are impressive considering how new the language implementations are, and *we argue that the results demonstrate that mature functional array languages have the potential to deliver performance competitive with the best available conventional techniques.* We look forward to seeing the developments in the area

in the coming years.

```
1  def relaxNAS (n:i64)
2                (elmAt:i32->i32->i32->real)
3                (ws: [4]real) : *[n][n][n]real =
4   let computeRow (ii: i64)(jj: i64) : [n]real =
5      let (i, j) = (i32.i64 ii, i32.i64 jj)
6      let f (k: i32) =
7         ( elmAt i ((j-1) % n) k +
8           elmAt i ((j+1) % n) k +
9           elmAt ((i-1) % n) j k +
10          elmAt ((i+1) % n) j k
11        ,
12          elmAt ((i-1) % n) ((j-1) % n) k +
13          elmAt ((i-1) % n) ((j+1) % n) k +
14          elmAt ((i+1) % n) ((j-1) % n) k +
15          elmAt ((i+1) % n) ((j+1) % n) k
16        )
17     let tab n h = map (h<-<i32.i64) [0...n-1]
18     let (u1s, u2s) = tab n f
19     let g u1s u2s (k: i32) = #[unsafe]
20           ws[0] * elmAt i j k
21         + ws[1] * (u1s[k] +
22                    elmAt i j ((k-1) % n) +
23                    elmAt i j ((k+1) % n))
24         + ws[2] * (u2s[k] + u1s[(k-1) % n] +
25                            u1s[(k+1) % n])
26         + ws[3] * (u2s[(k-1)%n] + u2s[(k+1)%n])
27     in  tab n g
28  in  map (\i -> map (computeRow i) (0...n-1))
29           (0...n-1)
30
```

Figure 2.4: A *simplified* implementation of the optimized computational kernel of MG, named `relaxNAS`, which is made generic by parameterizing it over an index function (`elmAt`) rather than a 3D array.

```
def getEachElm arr i j k =
  #[unsafe] arr[i, j, k]

def get8thElm arr i j k =
  if (i%2) + (j%2) + (k%2) == 3
  then #[unsafe]
       arr[i/2,j/2,k/2]
  else 0.0

def Q [n] relaxKer
          (zs: [n][n][n]real)
        : [2*n][2*n][2*n]real =
  relaxKer (2*n) (get8thElm zs)
           [1, 1/2, 1/4, 1/8]

def A [n] relaxKer
          (z: [n][n][n]real)
          (r: [n][n][n]real)
        : [n][n][n]real =
  relaxKer n (getEachElm z)
           [-8/3, 0, 1/6, 1/12]
  |> map2 (map2 (map2 (-))) r

def S [n] relaxKer
          (ws: [4]real)
          (r: [n][n][n]real)
          (z: [n][n][n]real)
        : [n][n][n]real =
  relaxKer n (getEachElm r) ws
  |> map2 (map2 (map2 (+))) z
```

Figure 2.5: The instantiations required to implement the computations $Q(\texttt{c2f}(zs))$, $r - A(z)$ and $z + S(r)$ in Algorithm 1. Of note, the actual implementation uses fast arithmetic for modulo operations. `#[unsafe]` prevents the insertion of dynamic bounds checks (which are expensive on GPUs).

```
double[d:shp] stencil(double[d:shp] x, double[d:wshp] w)
{
  return {i -> sum({j -> w[j]*x[mod(i+j-wshp/2,shp)]})
              | i < shp};
}
```

Figure 2.6: A Rank-Polymorphic stencil in SaC. For suitably chosen weights w, this computes the scaled (higher order) derivative of a discretized function x.

```
n0i, n0j, n0k = \
  (dace.symbol(s, dtype=dace.int32) for s in \
  ('n0i', 'n0j', 'n0k'))

@dace.program
def norm2u3_dace(r: dace.float64[n0i, n0j, n0k], \
                 dn: dace.int32):
    s = dace.reduce(lambda a, b: a + b,
                    r[1:-1, 1:-1, 1:-1]*r[1:-1, 1:-1, 1:-1],
                    identity=0.0)
    rnmu = dace.reduce(lambda a, b: max(a, b),
                       numpy.abs(r[1:-1, 1:-1, 1:-1]),
                       identity=0.0)
    rnm2 = numpy.sqrt(s / dn)
    return rnm2, rnmu

fast_sdfg = norm2u3_dace.to_sdfg()
auto_optimize(fast_sdfg)

faster_sdfg = norm2u3_dace.to_sdfg()
faster_sdfg.apply_transformations_repeated( \
    [MapReduceFusion])
tile_wcrs(faster_sdfg)
greedy_fuse(faster_sdfg)
```

Figure 2.7: DaCe Python implementation of norm2u3 and its optimisation.

**Algorithm 3** QuickHull

**Input:** $S$: a set of $n \geq 2$ two-dimensional points
**Output:** $CH$: the convex-hull set of $S$
$(A, B)$ = the leftmost and rightmost points of $S$
$S_{1,2}$ = points of $S$ above and below line $AB$
$CH = \{A, B\} \cup$ FINDHULL($S_1$, A, B) $\cup$ FINDHULL($S_2$, A, B)

**Algorithm 4** Divide-And-Conquer Helper

```
1:  procedure FINDHULL(S, P, Q)
2:      if S ≠ ∅ then
3:          C = furthest point of S from line PQ
4:          (S_l, S_r) = the points of S on the left-
5:                       and right-hand side of lines
6:                       CP and CQ, respectively
7:                       (and not inside ΔPCQ)
8:          Hull = {C} ∪
9:                 FINDHULL(S_l, P, C) ∪
10:                FINDHULL(S_r, C, Q)
11:     else
12:         Hull = ∅
13:     return Hull
```

**Algorithm 5** Standard Self-Attention

**Input:** $\mathbf{Q} : N \times d$ matrix of query weights.
**Input:** $\mathbf{K} : N \times d$ matrix of key weights.
**Input:** $\mathbf{V} : N \times d$ matrix of value weights.
**Output:** $\mathbf{O} : N \times d$ attention matrix.

1: $\mathbf{S} = \mathbf{Q} \times \mathbf{K}^T$
2: $\mathbf{P} = \mathit{softmax}(\mathbf{S})$
3: $\mathbf{O} = \mathbf{P} \times \mathbf{V}$

**Algorithm 6** Custom Attention

**Input:** same as Algorithm 5
**Output:** same as Algorithm 5

1: **for all** $i \in 0 \,..\, N-1$ **by** $d$ **do**
2: $\quad \mathbf{Qb} = \mathbf{Q}_{i:i+d,\,:}$
3: $\quad \mathbf{Sb} = \mathbf{Qb} \times \mathbf{K}^T$
4: $\quad \mathbf{Pb} = \mathit{softmaxOnline}(\mathbf{Sb})$
5: $\quad \mathbf{O}_{i:i+d,\,:} = \mathbf{Pb} \times \mathbf{V}$

Figure 2.8: In the code, $\mathbf{K}^T$ denotes the transpose of $\mathbf{K}$ and $\times$ denotes matrix multiplication. **forall** $i \in 0 \,..\, N-1$ **by** $d$ denotes a parallel (map-like) computation in which $i$ starts from 0 and advances with a step of $d$, i.e., $i = 0, d, 2 \cdot d, \ldots$. **Qb** iterates over the $d \times d$ slices of **Q** and the last line of Algorithm 6 computes the corresponding slice of the result **O**.

**Algorithm 7** Stable Softmax

**Input:** **S**: $M \times M$ input matrix.

1: **for all** $i \in 0 \,..\, M-1$ **do**
2:
3:
4:
5:
6: $\quad m_i = \overline{max}(S_{i,:})$
7: $\quad \tilde{p} : \text{vect}[M]$
8: $\quad$ **for all** $j \in 0 \,..\, M-1$ **do**
9: $\quad\quad \tilde{p}_j = \exp(S_{i,j} - m_i)$
10:
11: $\quad l_i = \overline{sum}(\tilde{p})$
12: $\quad$ **for all** $j \in 0 \,..\, M-1$ **do**
13: $\quad\quad P_{i,j} = \exp(S_{i,j} - m_i)/l_i$

**Algorithm 8** Online Softmax

**Output:** **P**: $M \times M$ output matrix.

1: **for all** $i \in 0 \,..\, M-1$ **do**
2: $\quad m_i = -\infty, l_i = 0$
3: $\quad$ **for** $jj \in 0 \,..\, M-1$ **by** $T$ **do**
4: $\quad\quad \tilde{s} : \text{vect}[T] = \mathit{copy}(S_{i,jj:jj+T})$
5: $\quad\quad m_i'' = \overline{max}(\tilde{s})$
6: $\quad\quad m_i' = \max(m_i,\ m_i'')$
7: $\quad\quad \tilde{p} : \text{vect}[T]$
8: $\quad\quad$ **for all** $j \in 0 \,..\, T-1$ **do**
9: $\quad\quad\quad \tilde{p}_j = \exp(\tilde{s}_j - m_i')$
10: $\quad\quad l_i' = \exp(m_i'' - m_i') \cdot \overline{sum}(\tilde{p})$
11: $\quad\quad l_i = l_i' + l_i \cdot \exp(m_i - m_i')$
12: $\quad\quad m_i = m_i'$
13: $\quad$ **for all** $j \in 0 \,..\, M-1$ **do**
14: $\quad\quad P_{i,j} = \exp(S_{i,j} - m_i)/l_i$

Figure 2.9: Algorithm 8 uses the same input and output as Algorithm 7. $\overline{max}$ and $\overline{sum}$ denote reductions of their vector argument. $\tilde{p}$ `: vect[X]` declares of a temporary vector $\tilde{p}$ of length $X$. **for all** denotes parallel (map-like) computations and **for** $jj \in 0 \,..\, M-1$ **by** $T$ denotes a sequential computation in which the loop index $jj$ advances with step $T$. Online Softmax is an algorithmic refinement that reorganizes the computation as a sequence of parallel computations of length $T$, where $T$ is a configurable tile size. This significantly reduces the number of accesses to global memory because vectors $\tilde{s}$ and $\tilde{p}$ now fit in scratchpad memory.

```
1  def mmmT [m][n][k] (as: [m][k]f32)(bs: [n][k]f32) :
2    [m][n]f32 =
3      map (\a -> map (\b -> map2 (*) a b |> f32.sum) bs) as
4  def mmm [m][n][k] (as: [m][k]f32) (bs: [k][n]f32) :
5    [m][n]f32 = mmmT as (transpose bs)
6
7  def FlashAttention [d][m](Q: [m][d][d]f32) (K: [m*d][d]f32)
8                            (V: [m*d][d]f32) : [m][d][d]f32 =
9    let mkOneBlock Qi =
10           let P = mmmT Qi K
11                 |> onlineSoftmax
12           in  mmm P V
13   in  map mkOneBlock Q
```

Figure 2.10: Custom Attention in Futhark. The implementation of onlineSoftmax (not shown) corresponds to Algorithm 8

```python
@dace.program
def flash_attention_dace(Q: dace.float32[N, d],
                         K: dace.float32[N, d],
                         V: dace.float32[N, d],
                         O: dace.float32[N, d]):

    for ti in dace.map[0:N:Ti]:

        m = np.full([Ti], -np.inf, Q.dtype)
        l = np.zeros([Ti], Q.dtype)
        S = np.empty([Ti, Tj], Q.dtype)
        Oi = np.zeros([Ti, d], Q.dtype)

        for tj in range(0, N, Tj):

            S[:] = Q[ti:ti+Ti, :] @ K[tj:tj+Tj, :].T

            max_row = np.max(S, axis=1)
            m_new = np.maximum(m, max_row)
            p_tilde = np.exp(S - m_new[:, np.newaxis])
            sum_row = np.sum(p_tilde, axis=1)
            l_tmp = l * np.exp(m - m_new)
            l_new = l_tmp + sum_row
            Oi[:] = (Oi * l_tmp[:, np.newaxis]
                     + p_tilde @ V[tj:tj+Tj, :])
                     / l_new[:, np.newaxis]
            m[:] = m_new
            l[:] = l_new

        O[ti:ti+Ti, :] = Oi
```

Figure 2.11: DaCe data-centric Python implementation of Flash Attention.

# Chapter 3

# Partitioning In-Place on Massively Parallel Architectures

## 3.1 Introduction

Partitioning is a fundamental operation for rearranging data. It takes an array of elements and a condition, permuting this array so all elements satisfying the condition are on the left of the array, and all elements that do not satisfy this criterion are on the right. More formally, for an array $X$ and a predicate pred, we compute a permutation $Y$ of $X$ and an index $m$ such that $\text{pred}(Y[i])$ for $i < m$ and $\neg\,\text{pred}(Y[i])$ for $i \geq m$. It can be used on its own in programs that classify data, but also as a subroutine in sorting [44] and computational geometry algorithms [172].

As this algorithm mainly moves data, it benefits from hardware that has high bandwidth. Graphics Processing Units (*GPUs*) are widely available processors and have much higher bandwidth than a CPU in the same price and power category, making them suitable candidates for partition algorithms.

These processors have such high bandwidth in part because the memory is physically attached to the processor. As a consequence, GPUs have less overall memory than comparable CPUs. This makes it important to use as little additional memory as possible, preferably a constant amount. In other words, GPU algorithms are ideally in-place, i.e., using $O(1)$ auxillary memory.

This chapter proposes an in-place partition algorithm for GPUs that is almost as fast as state-of-the-art out-of-place algorithms. Overall, we make the following contributions.

- We describe an in-place parallel algorithm for partitioning that maps well on GPUs.
- We prove our algorithm has at most $3n+O(1)$ data movements for any input of size $n$, and approximately $2n$ for random input.

- We evaluate our implementation on two GPUs for a range of inputs.

## 3.2 Background

### Graphics Processing Units

A GPU is a specialized processor that emphasizes execution units at the cost of control logic. To decrease demand on the control logic, the programmer needs to have groups of threads do the same instruction (*SIMT*) for optimal performance. For similar reasons, the threads must access memory in specific patterns to make full use of the bandwidth. This is called *coalesced* access.

A GPU has different levels of parallelism that allow for different levels of flexibility. On the lowest and least flexible level are groups of usually 32 threads (*warps*) that must be programmed SIMT style. These warps are grouped in *thread blocks* that are mapped on hardware units called *streaming multiprocessors (SMs)*, sharing some fast memory. We map multiple of these groups on one SM so the SM can hide latency by switching between thread blocks.

### Distributed-Memory Computing

If a data structure is too large to fit on one physical machine, it has to be distributed over a network of processors, each with their own memory. Coordinating such a network to collaboratively compute some task is called *distributed-memory computing*. This is relevant to this chapter as we will look at a GPU as a distributed system.

Following the notational convention of [27], we write $p$ for the total number of processors and $P(s)$ for the processor with index $0 \leq s < p$. A division of a large data structure $X$ into a collection of smaller data structures $\{X^{(s)} : 0 \leq s < p\}$ and a correspondence between them is called a *distribution*. The subset $X^{(s)}$ is local to processor $P(s)$. Figure 3.1 illustrates some examples of distributions. If $X$ has index set $I = \{0, \cdots, n-1\}$, we define the block-cyclic distribution with block-size $b$ over $p$ processors as

$$I^{(s)} = \{(k \cdot p + s) \cdot b + j : 0 \leq j < b,\ 0 \leq (k \cdot p + s) \cdot b + j < n\},$$

where $X^{(s)} := \{X[i] : i \in I^{(s)}\}$. The block distribution is one extreme of this, with $b = \left\lceil \frac{n}{p} \right\rceil$, and the cyclic distribution the other extreme with $b = 1$. It will be convenient to use interval notation with global indices: $X^{(s)}[a, b) := \{X[i] : i \in I^{(s)}, \quad a \leq i < b\}$.

Using a distributed data structure requires data movement between machines for accessing non-local data. This is an expensive operation, so designing a fast distributed algorithm often boils down to choosing the distribution in a way that minimizes data movement.

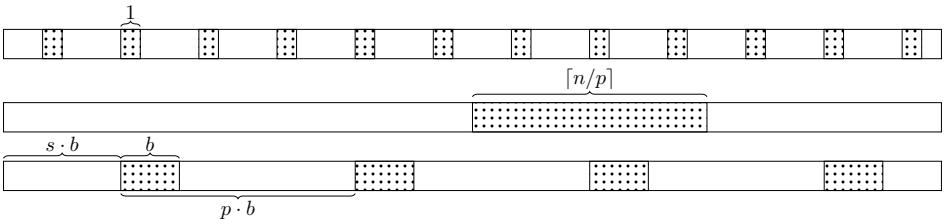



Figure 3.1: Three distributions of $n$ elements over $p$ processors. The dotted region is the subarray $X^{(s)}$ that belongs to $P(s)$. From top to bottom: cyclic, block, block-cyclic (with block size $b$).

### Computational Model

We use the following mental model: we consider the GPU a distributed computer, where the processors are GPU thread blocks. Within that model, each processor is itself a parallel computer that uses the SIMT paradigm.

We model the cost of an algorithm by data movement, more specifically reads and writes to global memory. The cost of computation and the cost of data movement in local memory is not significant for algorithms with low algorithmic complexity.

In contrast to clusters, most GPUs are not partially allocated. For this reason, we consider the number of processors constant, and aim to expose sufficient concurrency to fully saturate the hardware, rather than maximizing scalability of the algorithm.

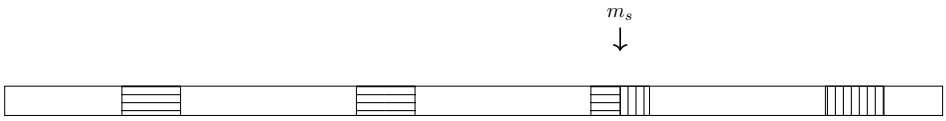
## 3.3 Partition Algorithm

The main idea behind the algorithm is to minimise overall data movement by choosing a suitable distribution. The algorithm consists of two phases: a *local partition* where each processor $P(s)$ partitions its local data $X^{(s)}$ without coordinating with the other processors, and then a *cleanup phase* where we correct any mistakes. Figure 3.2 illustrates the result of the local partition using a block-cyclic distribution. Each processor permutes its $X^{(s)}$ and computes a local split $m_s$ such that $X^{(s)}[0, m_s)$ satisfies pred and $X^{(s)}[m_s, n)$ does not. The global split $m$ is equal to $\sum_{s=0}^{p-1} |X^{(s)}[0, m_s)|$. This divides the incorrect points into two sets: a set $L$ that should be moved to the left, and a set $R$ that should be moved to the right, or more precisely

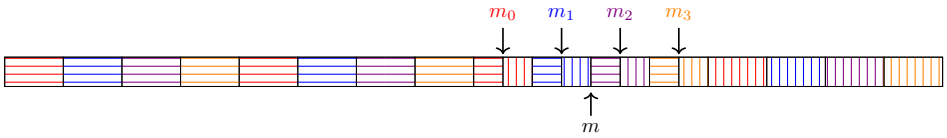
$$L := \bigcup_{s \in J} X^{(s)}[m, m_s), \quad R := \bigcup_{s \notin J} X^{(s)}[m_s, m), \quad J := \{s : m_s > m\}.$$

The cleanup phase swaps $L$ and $R$.

Note here that all processors, i.e., all GPU thread blocks can be executed independently during the local partition phase, and that a block-cyclic distribution of input on reasonably distributed input will lead to a reasonably small sets $L$ and




(a) Local partition of $X^{(s)}$, SPMD view from $P(s)$




(b) Local partition for four processors, depicted in different colours.

Figure 3.2: Local partition of each subarray $X^{(s)}$. The horizontal lines satisfy pred, while the vertical do not.

$R$ relative to the overall size of the input. The two remaining challenges are to use individual GPU thread blocks to partition their corresponding local array $X^{(s)}$ effectively in-place, and to efficiently implement the cleanup phase.

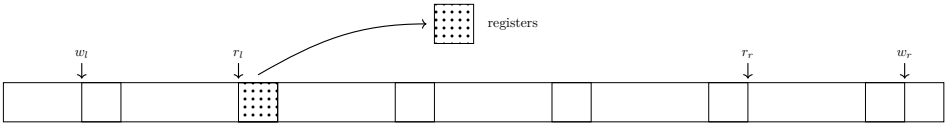
## Local Partition

To use all parallelism within a GPU thread block, we in principle want to apply the state-of-the-art algorithm for GPUs—a scan-based implementation [173]—on the local subarrays $X^{(s)}$. However, this is not in-place. So instead of running this algorithm on all of $X^{(s)}$, we cut it into chunks of a fixed size which we process one after the other. While this limits the available parallelism, it ensures that the space overhead is independent of the input size, rendering our algorithm an in-place variant. Since we have limited the physically available parallelism to individual GPU thread blocks, the limited parallelism does not impede on the overall performance.

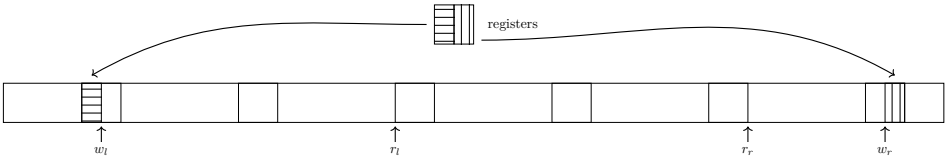
We illustrate this algorithm in Figure 3.3. The key idea is to maintain read and write pointers for the left and right side of the array, $r_l$, $w_l$, $w_r$, $r_r$. At any time, we maintain the following invariants:

- we have $\text{pred}(X^{(s)}[0, w_l))$,
- we have $\neg\,\text{pred}(X^{(s)}[w_r, n))$,
- we buffer $X^{(s)}[w_l, r_l)$ and $X^{(s)}[r_r, w_r)$, and
- the data in $X^{(s)}[r_l, r_r)$ is not partitioned yet.

At the start of the algorithm, we initialise $w_l$, $r_l$ to the first index of $X^{(s)}$, $w_r$, $r_r$ to one past the last index of $X^{(s)}$ and then establish the invariant by buffering $b$ elements at $r_l$, $r_r$ and advancing these pointers. We then repeatedly read in $b$ elements either starting at $r_l$ or ending at $r_r$, partition it using the scan-based algorithm, write it back to $w_l$, $w_r$, and update the pointers. The invariants are maintained if we read at $r_l$ when $|X^{(s)}[w_l, r_l)| \geq |X^{(s)}[r_r, w_r)|$.




(a) Step 1: load a block into registers from either the left or right side.




(b) Step 2: partition, write elements to $w_l$, $w_r$, and update pointers.

Figure 3.3: An iteration of the local partition algorithm where we read from the left.

## Cleanup

The most straightforward way of swapping $L$ and $R$ is to split each into $p$ chunks of roughly equal size and then make each processor swap one of these chunks. The usual way of partitioning a set $Y$ into $p$ chunks is to take chunksize $C = \lceil |Y|/p \rceil$, and then give indices $[sC, \min((s+1)C, n))$ to processor $P(s)$. Unfortunately, there is no straightforward way to index $L$ and $R$ from 0 to $|L|$, so we have to do some work.

We used an index set $J := \{s : m_s > m\}$ to express $L$ and $R$. We can test whether $s \in J$ by computing $w_s = |X^{(s)}[0, m)| - |X^{(s)}[0, m_s)|$. This is negative if and only if $s \in J$. Furthermore, if $s \in J$, then $w_s = -|X^{(s)}[m, m_s)|$ and if $s \notin J$, then $w_s = |X^{(s)}[m_s, m)|$. We then partition $\{w_s : 0 \leq s < p\}$ into negative and non-negative values, yielding permutation $\sigma$ and a split $\alpha$ such that $w_{\sigma(s)} < 0$ if and only if $s < \alpha$. This permutation lets us to express $L$ and $R$ as

$$L = \bigcup_{s=0}^{\alpha-1} X^{(\sigma(s))}[m, m_{\sigma(s)}), \quad R = \bigcup_{s=\alpha}^{p-1} X^{(\sigma(s))}[m_{\sigma(s)}, m).$$

We can now define an order on $L$ as follows: let $j_s$ be the $j$th element in $X^{(\sigma(s))}[m, m_{\sigma(s)})$, and $k_t$ the $k$th element in $X^{(\sigma(t))}[m, m_{\sigma(t)})$. Then $j_s < k_t$ if $s < t$ or $s = t$ and $j < k$. The case $R$ is analogous, except we take the $j$th element in $X^{(\sigma(s))}[m_{\sigma(s)}, m)$ instead of $X^{(\sigma(s))}[m, m_{\sigma(s)})$. We illustrate this in Figure 3.4.

To determine where $s$th chunk of $L$ starts, we must find the processor index $t$ and local offset $j$ such that the $s{\cdot}C$th element of $L$ is the $j$th element of $X^{(t)}[m, m_t)$. As the index of the first element in $X^{(\sigma(t))}[m_{\sigma(t)}, m)$ is $\sum_{i=0}^{t-1} |w_{\sigma(i)}|$, index $sC$ falls in $X^{(\sigma(t))}[m_t, m)$ whenever

$$\sum_{i=0}^{t-1} |w_{\sigma(i)}| \leq sC < \sum_{i=0}^{t} |w_{\sigma(i)}|.$$

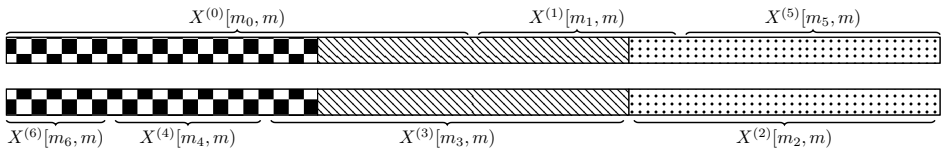



Figure 3.4: Example of $L$ and $R$ for $w = [-9, -4, 6, 7, 3, -5, 2]$, and permutation $\sigma = [0, 1, 5, 6, 4, 3, 2]$. The split $\alpha$ is 3. The patterns divide the arrays in three chunks. (The actual algorithm would use $p = 7$ chunks.)

This is equivalent to

$$\left\lceil \sum_{i=0}^{t-1} |w_{\sigma(i)}|/C \right\rceil \leq s < \left\lceil \sum_{i=0}^{t} |w_{\sigma(i)}|/C \right\rceil,$$

which can be efficiently computed. The local offset of $sC$ is then $sC - \sum_{i=0}^{t-1} |w_{\sigma(i)}|$. The case $R$ is analogous.

## Data Movement Analysis

The local partition always reads and writes each element, resulting in $2n$ data movement where $n$ is size of $X$. After the local partition, we have to move only the incorrect elements in the cleanup phase. Therefore, it suffices to consider the number of incorrect elements after the local partition to derive bounds on the total data movement. We derive a worst-case bound in Theorem 1, and a probabilistic bound in Theorem 2.

### Worst-case Input

A GPU has a fixed number of SMs and computational resources per SM, making the choice of $p$ thread blocks and $b$ threads per block independent of the input. For this reason, the following theorem states that the local partition can place at most $n/2 + O(1)$ elements incorrectly.

**Theorem 1.** *After the local partition, at most $n/2 + pb$ elements are placed incorrectly.*

*Proof.* The crucial idea behind the proof is that $|L| = |R|$, so we do not have to bound $|L|$ and $|R|$, but only $\min(|L|, |R|)$. We can express upper bounds for $|L|$ and $|R|$ in terms of $m$ and the number of processors that have an excess of elements to the left. If this bound is increasing in either variable for $|L|$, it is decreasing for $|R|$, and the other way around. This means either upper bound can be large, but not both of them at the same time, so the bound on $\min(|L|, |R|)$ is small.

For $s \in J$, the elements in $X^{(s)}[0, m)$ are placed correctly, and for $s \notin J$, the elements in $X^{(s)}[m, n)$ are correct. This gives us a lower bound on the correct elements before $m$

$$\sum_{s \in J} |X^{(s)}[0, m)| > |J| \cdot \left( \frac{m}{p} - b \right).$$

Hence

$$|R| < m - |J| \cdot \left(\frac{m}{p} - b\right) = \left(1 - \frac{|J|}{p}\right) \cdot m + |J|b.$$

Similarly,

$$\sum_{s \notin J} |X^{(s)}[m,n)| > (p - |J|) \cdot \left(\frac{n-m}{p} - b\right),$$

so

$$|L| < n - m - (p - |J|) \cdot \left(\frac{n-m}{p} - b\right) = \frac{|J|}{p} \cdot (n-m) + (p - |J|)b.$$

As $|L| = |R|$, we have

$$\begin{aligned} |L| &< \min\left(\frac{|J|}{p} \cdot (n-m) + (p-|J|)b, (1 - \frac{|J|}{p}) \cdot m + |J|b\right) \\ &\leq \min\left(\frac{|J|}{p} \cdot (n-m), \left(1 - \frac{|J|}{p}\right) \cdot m\right) + \min((p-|J|)b, |J|b). \end{aligned}$$

Considering $|J|/p$ a variable, the expression $|J|/p \cdot (n-m)$ is an ascending line, and $(1 - |J|/p) \cdot m$ a descending. So the minimum is the intersection of the two lines, which gives $|J|/p = m/n$. Substituting this value obtains upper bound $m/n \cdot (n-m)$. Using calculus we find this is at most $n/4$ (for $m = n/2$). Analogously, we have $\min((p-|J|)b, |J|b) \leq p/2$. We conclude $|L| < n/4 + pb/2$.

Using $|L| = |R|$ we derive the final upper bound on the number of incorrect elements $|L| + |R| = 2|L| < n/2 + pb$. □

### Random Input

The intuition behind the analysis on random inputs is that the expected value of the local splits $m_s$ will be around index $\mu n$, where $\mu$ is the probability that the predicate is true. This is also the expected value of the global split $m$. We can then use strong tail bounds on $\mathbb{P}(|Z - \mathbb{E}[Z]| > k)$ for suitable random variables $Z$. This is responsible for the $\epsilon n$ and exponentially decaying probability in Theorem 2. We then bound the difference between the expected value of $m_s$ and $\mu n$, which costs us the additional summand $2pb$. We will need the following lemma to combine bounds.

**Lemma 1.** *Let $Z_1, \cdots, Z_l$ be (not necessarily independent) random variables $\Omega \to \mathbb{R}$. Then*

$$\mathbb{P}\left(\left|\sum_{i=1}^{l} Z_i\right| > k\right) \leq \sum_{i=1}^{l} \mathbb{P}\left(|Z_i| > \frac{k}{l}\right).$$

*Proof.* By triangle inequality

$$\left|\sum_{i=1}^{l} Z_i\right| \leq \sum_{i=1}^{l} |Z_i|,$$

so $k < \left|\sum_{i=1}^{l} Z_i\right|$ implies there must be some $Z_i > k/l$. In other words,

$$\left\{\omega \in \Omega : \left|\sum_{i=1}^{l} Z_i(\omega)\right| > k\right\} \subseteq \bigcup_{i=1}^{l} \left\{\omega \in \Omega : |Z_i(\omega)| > \frac{k}{l}\right\}.$$

The lemma now follows from subadditivity. □

**Theorem 2.** *For i.i.d. random inputs $X[0], \cdots, X[n-1]$ or a randomly permuted input, we have to accept that $2pb$ elements may be in incorrect position after the local partition. However, it is extremely unlikely that the fraction $\epsilon$ of incorrect elements is much larger than $O(\sqrt{p/n})$. To be more precise, for $I$ the set of incorrect points:*

$$\mathbb{P}\left(|I| > 2pb + \epsilon n\right) \leq 4p \exp\left(-\frac{\epsilon^2 n}{3p}\right).$$

**Example 1.** *The default values of our algorithm are $p = 4096$, $b = 512$. Suppose we partition an array of 8 GB of doubles, so $n = 10^9$. The bound starts being less than 100% when we look at more than 1.1% of values being incorrect ($\epsilon = 0.011$). The probability that more than 1.1% of values are incorrect is at most 87%. The probability that more than 1.3% are wrong is no more than 1.7%, and the probability that more than 2% of values are wrong is at most $3 \cdot 10^{-10}$%.*

*Proof.* The number of incorrect elements is

$$|I| = \sum_{s=0}^{p-1} \left||X^{(s)}[0, m)| - |X^{(s)}[0, m_s)|\right|.$$

We can express these random variables with indicator functions

$$Y_i(\omega) = \begin{cases} 1 & \text{if } \operatorname{pred}(X_i) \\ 0 & \text{if } \neg \operatorname{pred}(X_i) \end{cases},$$

which gives

$$m = \sum_{i=0}^{n-1} Y_i, \quad C_s := |X^{(s)}[0, m_s)| = \sum_{i \in I^{(s)}} Y_i$$
$$D_s := |X^{(s)}[0, m)| = b \cdot \left\lfloor \frac{m}{pb} \right\rfloor + m \mod b.$$

For both i.i.d. random input and randomly permuted input, the indicator functions $Y_i$ are i.i.d. Bernoulli random variables.

With this notation, the applications of Lemma 1 give

$$\mathbb{P}\left(|L| + |R| > 2pb + \epsilon n\right) \leq \sum_{s=0}^{p-1} \mathbb{P}(|D_s - C_s| > 2b + \epsilon n/p). \tag{3.1}$$

Both $C_s$ and $m$ are binomial distributions, with success rate $\mu = \mathbb{E}[Y_i]$, which means we can apply Hoeffding's inequality on them. To that end, we use $||I^{(s)}| - n/p| \leq b$ and $|D_s - m/p| \leq b$ to obtain upper bound

$$\begin{aligned}
|D_s - C_s| &= \left| \left(\frac{m}{p} - \frac{n}{p}\mu\right) + (|I^{(s)}|\mu - C_s) + D_s - \frac{m}{p} + \frac{n}{p}\mu - |I^{(s)}|\mu \right| \\
&\leq \left| \left(\frac{m}{p} - \frac{n}{p}\mu\right) + (|I^{(s)}|\mu - C_s) \right| + b + \mu b \\
&\leq \left| \left(\frac{m}{p} - \frac{n}{p}\mu\right) + (|I^{(s)}|\mu - C_s) \right| + 2b.
\end{aligned}$$

We conclude that $|D_s - C_s| > 2b + \epsilon n/p$ implies

$$\left| \left(\frac{m}{p} - \frac{n}{p\mu}\right) + \left(|I^{(s)}|\mu - C_s\right) \right| > \frac{\epsilon n}{p}.$$

Again by Lemma 1, we have

$$\mathbb{P}(|D_s - C_s| > 2b + \epsilon\frac{n}{p}) \leq \mathbb{P}\left(|\frac{m}{p} - \frac{n}{p}\mu| > \epsilon\frac{n}{2p}\right) + \mathbb{P}\left(||I^{(s)}|\mu - C_s| > \epsilon\frac{n}{2p}\right).$$

Both $m$ and $C_s$ are sums of independent random variables, so by Hoeffding's inequality, we have

$$\mathbb{P}\left(||I^{(s)}|\mu - C_s| > \frac{\epsilon n}{2p}\right) \leq 2\exp\left(-\frac{\epsilon^2 n^2}{2p^2|I^{(s)}|}\right).$$

As $2p^2|I^{(s)}| \leq 2p^2(n/p + b) = 2pn + p^2 b \leq 2pn + pn = 3pn$, we find upper bound

$$\mathbb{P}\left(\left||I^{(s)}|\mu - C_s\right| > \frac{\epsilon n}{2p}\right) \leq 2\exp\left(-\frac{\epsilon^2 n}{3p}\right). \tag{3.2}$$

Also by Hoeffding's inequality, we have

$$\mathbb{P}\left(|\frac{m}{p} - \frac{n}{p}\mu| > \frac{\epsilon n}{2p}\right) = \mathbb{P}\left(|m - n\mu| > \frac{1}{2}\epsilon n\right) \leq 2\exp\left(-\frac{1}{2}\epsilon^2 n\right). \tag{3.3}$$

The theorem follows by combining inequalities 3.1, 3.2, and 3.3. □

## 3.4 Implementation

We implement this algorithm in CUDA, using the CUB library [187]. This library contains implementations of several building blocks, implemented on the level of warps, thread blocks, and the entire device. The code [144] and instructions for reproducing our experiments are available on Zenodo.

We do the scan-based partition on the level of thread blocks. By default, we use 128 threads per block and 4 items per thread, which gives block parameter $b = 128\cdot$

$4 = 512$. These values are C++ template parameters and can be changed by the user. CUB contains a scan and scatter function, which we use to straightforwardly build the local partition.

The value for $p$ is also a template parameter. The optimal choice of $p$ would be the number of SMs multiplied by the number of blocks that can be mapped to each SM concurrently (the *occupancy*) if these would be optimally scheduled. However, our experiments indicated that the scheduling is not always optimal, and this parameter would have to be changed for each architecture. So long as $pb \ll n$, it does not hurt to take $p$ big. For this reason we have chosen default 4096.

The partition in the cleanup phase is small, so we use the out-of-place implementation of CUB to compute $\sigma$. We also use CUB's scan function to compute $\sum_{i=0}^{t} |w_{\sigma(i)}|$ for $t = 1, \cdots, p$.

## 3.5 Performance Evaluation

As partitioning has a small amount of work per element, the performance will be limited by bandwidth. We can evaluate the performance of our algorithm by comparing the achieved bandwidth with the peak bandwidth the GPU is capable of. We estimate this peak bandwidth by using one of the STREAM [168] benchmarks, a collection of simple computations that stresses the memory subsystem of a computer. To most closely match the structure of a partition, we have chosen for SCALE. This operation multiplies all elements of an array by a fixed constant $\alpha$ (we have chosen $\alpha = 2$).

### Methodology

We use a consumer GPU (3070 Ti) and two server GPUs (A30, H100) for our measurements, with 8 GiB, 24 GiB, 80 GiB of VRAM. We use NVCC versions 12.6.77, 12.6.68, 12.6.20, and CUB versions 1.017.02, 2.005.00, 2.005.00.

The CUDA compiler NVCC generates an intermediate representation which is compiled to machine code by the driver just before execution. After the first execution, the compiled machine code can be fetched from an instruction cache. Furthermore, the clock frequency is not constant. For this reason, we do some untimed warmup runs before our measurements. We call the partition function in a loop until 500 GB of data has been moved.

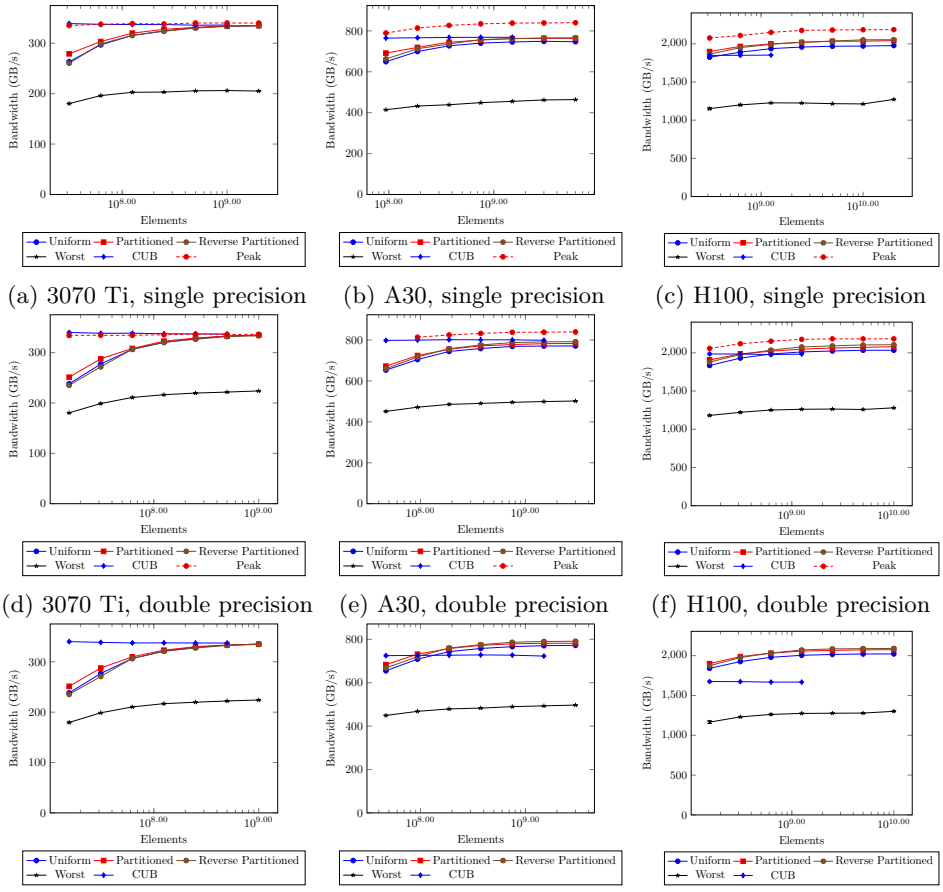
We repeat each of these experiments ten times and report the mean and standard deviation through error bars.

### Discussion

We investigate two variables for each machine and data type: input size, and input structure. This is graphed in Figure 3.5, where we use a logarithmic scale on the $x$-axis with each point a factor two apart. We investigate data types float, double, and struct { int x; int y; }. We construct an input that gets within $pb$ of the worst case of Theorem 1 by setting $X^{(s)}$ to pred for even $s$, and to $\neg$ pred for odd $s$. As we have chosen $p$ even, the local partition will place $n/2$ elements

incorrectly. For random input we generate values such that $\mathbb{P}(\text{pred}(X[i])) = 0.5$. We also generate a structured input that is already partitioned, and one that is reversed partitioned, e.g. partitioned according to $\neg\,\text{pred}$. For CUB the results are within a standard deviation for all input structures, so we only graph one.

We do all our measurements on a range of sizes. Our algorithm uses arrays of 8 GB, 24 GB, 80 GB respectively, or about 93 % of the available VRAM. CUB can partition arrays half as large. There is also a maximum of $2^{31} - 1$ elements because CUB uses an int type internally for offsets.




(a) 3070 Ti, single precision (b) A30, single precision (c) H100, single precision

(d) 3070 Ti, double precision (e) A30, double precision (f) H100, double precision

(g) 3070 Ti, struct of int (h) A30, struct of int (i) H100, struct of int

Figure 3.5: Bandwidth partition algorithms

There are three types of overhead that explain why our algorithm does not attain the performance of STREAM.

First, the local partition and index arithmetic of block-cyclic arrays increase the latency between load and stores. This limits the performance of our algorithm to 89 %–99 % of STREAM on large, non-adversarial arrays. There is no significant

difference between the reverse partitioned and uniform input structures, indicating that the local partition placed most points correctly. This agrees with Theorem 2. CUB also needs to do local partitioning, but does not need index arithmetic, yielding slightly better performance on float and double: 92 %–99 % of STREAM. For the struct of int the performance drops to as little as 76 % of our performance. We were unable to pinpoint the exact cause, but believe the more complicated data type inhibits optimising one of the many C++ abstractions CUB uses.

Second, our algorithm has significant constant overhead. We perform ten kernel calls as opposed to CUB's one, and the cleanup phase also has large constants. This causes a noticeable drop in performance for small arrays, but the performance is still at least 79 % of STREAM's for arrays of non-adversarial input. It is interesting that CUB is slightly faster than STREAM on the 3070 Ti, and that the already partitioned input is slightly faster than the reverse partitioned input. This suggests that writing back identical memory is slightly faster for small arrays.

Third, the cleanup phase moves a significant amount of data for adversarial arrays. We expect the performance for the worst-case input to be roughly linear in data movement, so this bandwidth should be a fraction $2n/(3n) \approx 0.67$ of random input. This is also observed in practice, where the bandwidth of worst-case input is 0.63 %–0.67 % of the bandwidth of random input.

## 3.6 Related and Future Work

The main structure of a local partition and a cleanup phase resembles an algorithm by Francis et al. [79]. The difference is that we use a block-cyclic distribution instead of a cyclic distribution, and have a parallel cleanup phase, instead of partitioning $[\min_s m_s, \max_s m_s)$ sequentially.

The local partition can be seen as the GPU analogue of an AVX-512 partition algorithm [38]. A thread-block replaces the CPU core, and a scan-based partition algorithm [32] within that thread block replaces the AVX-512 partition instruction.

Gu et al. use a more abstract model of parallel in-place algorithms [98] based on the work-span model [215]. This algorithm is work-efficient and makes a trade-off based on a parameter $\epsilon$ between $O(n^\epsilon \log(n))$ span and $O(n^{1-\epsilon})$ auxilary space. Their implementation targets multicore machines instead of GPUs.

Partitioning is usually not used on its own, so we will mention some motivating applications. A classification algorithm such as the Support Vector Machine (SVM) [57] finds a hyperplane separating points in $\mathbb{R}^n$. To compute efficiently on these two sets, it can be useful to partition them in memory, where pred is their position relative to the hyperplane. The same predicate is used in computational geometry, where the Quickhull [37] algorithm recursively partitions points based on their orientation to a hyperplanes. Not all algorithms have fixed predicates. The sorting algorithm Quicksort [110] can choose any element as pivot.

These applications suggest opportunities for future work. The SVM algorithm can be extended to multiclass SVM which partitions data in more than two classes. Similarly, sorting algorithms aim to reduce the number of passes over the input by partitioning in multiple classes as well. One of these algorithms, IPS4o [12],

is parallel and in-place. It maps well to CPUs, but to map well to GPUs a SIMD/SIMT approach to filling the bins is needed.

## 3.7 Conclusion

We have designed an in-place partition algorithm and implemented it for GPUs. This algorithm can handle cases that are twice as large as that of the state-of-the-art CUB implementation. For the cases we expect users to be most interested in — large, non-adversarial arrays — we obtain 97 %–100 % of CUB's performance and 89 %–99 % of the hardware's peak. The limitations of our work are small arrays or arrays with an adversarial structure, where the performance may drop to 53 %. We have proven a worst-case bound on data movement, and a probabilistic bound for random input.

We viewed the GPU as a distributed computer by looking at thread blocks as nodes and threads within a block as SIMD lanes. This allows algorithms from the literature to be reinterpreted, serving as starting point for this algorithm. It also simplifies the description and analysis.

# Chapter 4

# VQhull: a Fast Planar Quickhull

## 4.1 Introduction

The convex hull of a set of points is the smallest superset closed under taking convex combinations, a type of weighted average. The convex hull of a finite set in 2D is a polygon, which can be described by its vertices. Finding these vertices and listing them in clockwise order is a fundamental algorithm in computational geometry. Some problems make use of the convex hull construction directly, while others use a convex hull algorithm as subroutine. Applications can be found in a wide range of domains, such as collision detection [227], path planning [171], and urban planning [131].

There exists a multitude of algorithms for finding the convex hull [92, 125, 70, 196, 42, 4, 10, 53, 37, 48]. An implementation of Quickhull is faster than other algorithms on most data sets, but relating the runtime performance to the peak performance of the hardware shows there is still significant room for improvement. In this chapter we aim to close this gap by providing a novel parallelisation of the Quickhull algorithm called VQhull.

Obtaining good performance on a multicore machine is a matter of exposing sufficient parallelism and minimising data movement. VQhull uses parallelism by both vector instructions and multiple CPU cores. We need to make minor algorithmic adjustments to make effective use of the memory subsystem.

The ICT sector is currently estimated to account for 2.1 %–3.9 % of global emissions — more than the aviation industry [80]. While performance analyses usually focus only on runtime, it is clear that reducing the energy consumption of software is crucial for reducing greenhouse gas emissions of ICT. We describe our method for measuring energy consumption, using which we evaluate the energy-efficiency of our approach.

We make the following contributions.

- We present a vectorized algorithm for extracting two disjoint subsets in-

place. (Section 4.3)

- We implement VQhull with a parallelized subset extraction over multiple cores in a bandwidth-friendly manner. (Section 4.4)
- We provide an extensive runtime performance and energy consumption analysis of our approach compared to the state of the art, on three platforms. (Section 4.5)

## 4.2 Background

### Quickhull Algorithm

The Quickhull algorithm [37] is a widely used technique for finding the convex hull of a finite point set. It operates similarly to the QuickSort algorithm, as both apply a divide and conquer approach to recursively partition a set.

Figure 4.1 illustrates the basic idea behind Quickhull—Algorithm 9. If we have three points $p$, $r$, $q$ on the convex hull of $P$, then the points inside that triangle are not in the convex hull and can be discarded. The points $S_1$ to the left of $\overrightarrow{pr}$, and the points $S_2$ to the left of $\overrightarrow{rq}$ may be on the convex hull, and are recursively inspected.

To find three points on the convex hull, we can start with the leftmost and rightmost points $p$ and $q$. We use the degenerate triangle $\Delta pqp$ to bootstrap the algorithm. The arguments of the recursive function `Hull` give two points on the convex hulls of $S_1$, $S_2$. To find the third point we use the following observation. If $x$, $y$ is on $CH(P)$, then a point $z \in P$ with maximum distance to $\overrightarrow{xy}$ must also be on the convex hull.

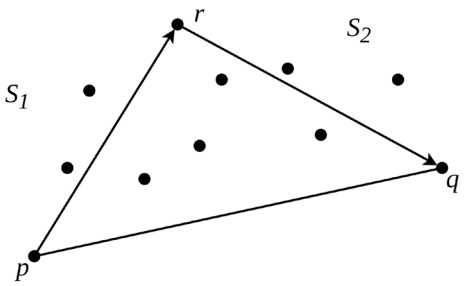



Figure 4.1: An example of the first partitioning step of the Quickhull algorithm.

We can efficiently and accurately tell whether $u$ is to the left of $\overrightarrow{pq}$ by testing

$$(p_x - u_x) \cdot (q_y - u_y) > (p_y - u_y) \cdot (q_x - u_x),$$

and check whether $u$ is farther than $u'$ from $\overrightarrow{pq}$ by testing

$$(q_y - p_y)(u_x - u'_x) < (q_x - p_x)(u_y - u'_y).$$

**Algorithm 9** Quickhull algorithm.

**Input:** The set of points $P \subset \mathbb{R}^2$.
**Output:** The vertices $H \subseteq P$ of the convex hull in clockwise order.

Let $p \in P$ be the leftmost point.
Let $q \in P$ be the rightmost point.
$S_1 := \{u \in P \mid u \text{ to the left of } \overrightarrow{pq}\}$
$r_1 := \operatorname{argmax}_{u \in S_1} d(u, \overrightarrow{pq})$
$S_2 := \{u \in P \mid u \text{ to the right of } \overrightarrow{pq}\}$
$r_2 := \operatorname{argmax}_{u \in S_2} d(u, \overrightarrow{pq})$
$H := \{p\} \cup \{q\} \cup \textsc{Hull}(S_1, p, r_1, q) \cup \textsc{Hull}(S_2, q, r_2, p)$
**function** $\textsc{Hull}(P, p, r, q)$
    **if** $|P| \leq 1$ **then**
        **return** $P$
    **else**
        $S_1 := \{u \in P \mid u \text{ to the left of } \overrightarrow{pr}\}$
        $r_1 := \operatorname{argmax}_{u \in S_1} d(u, \overrightarrow{pr})$
        $S_2 := \{u \in P \mid u \text{ to the left of } \overrightarrow{rq}\}$
        $r_2 := \operatorname{argmax}_{u \in S_2} d(u, \overrightarrow{rq})$
        **return** $\{r\} \cup \textsc{Hull}(S_1, p, r_1, r) \cup \textsc{Hull}(S_2, r, r_2, q)$

The partitioning step is then recursively applied in Algorithm 9 on the remaining subsets $S_1$ and $S_2$. Each call adds the element of its subset with maximum distance to $\overrightarrow{xy}$ to the convex hull.

## Performance on Central Processing Units

Implementation-related performance improvement comes mainly from exposing parallelism and reducing data movement. We review what this looks like on the hardware we target, a Central Processing Unit (*CPU*), so we can design the parallel algorithm with this in mind.

### Branch Prediction

A CPU has multiple execution units per core. If there are no dependencies between instructions, these units can execute instructions in parallel (*instruction level parallelism*), benefiting the performance. To keep a steady flow of instructions, the CPU will make a guess when encountering a conditional jump—generated by if statements and loops in high level languages. This is called branch prediction and improves performance when guessed correctly. If the guess is incorrect, the CPU must make an expensive recovery.

#### Vectorisation

A *vector* or *SIMD* instruction is one that operates on multiple primitive types (such as integers or floating point numbers) at the same time. The number of elements is called the *vector width*, and is typically between 2 and 8 elements.

These instructions can be used to do multiple arithmetic operations such as a[0] = b[0] + c[0]; a[1] = b[1] + c[1]; in one instruction, but also data-movement such as a[0] = flag[0] ? b[0] : c[0]; a[1] = flag[1] ? b[1] : c[1];. This reduces the number of instructions necessary, and instructions that permute data can also be used instead of branches.

#### Data Movement and Multithreading

This parallelism increases the number of computations we can do in a given interval, but that is not the only limiting factor. After all, we also need to load and store data fast enough. We call applications limited by the number of computations we can do *compute-bound*, and applications limited by data movement *memory-bound*.

Branch prediction and vectorisation happens on each core. Similarly, each core also has some fast local memory called a cache. These resources scale with using more cores. The main memory however, shares a common data bus with all cores. For this reason problems are more likely to become memory-bound when using more cores.

## 4.3 Subset Extraction

The main computation of Algorithm 9 is to extract subsets $S_1$, $S_2$ from $P$. This resembles the partition step in QuickSort, except that we discard more points. This makes it more like a Dutch National Flag (DNF) problem, where we want to extract three sets: $S_1$, $S_2$, and $P \backslash (S_1 \cup S_2)$. We can implement this DNF-like problem more efficiently by exploiting that we do not need to preserve the last set. We first describe a vectorized algorithm for a single core, and then parallelise it over multiple cores. Though the algorithm on one core is parallel in its use of SIMD instructions and out-of-order execution, we follow convention and call it sequential.

### Sequential Extraction

The main idea behind the extraction on a single CPU core is to classify multiple points simultaneously by using vector instructions. These registers can hold multiple elements, and corresponding instructions can operate on all of them simultaneously.

The main instruction used is a compression, vcompresspd, from the Intel AVX-512 instruction set, illustrated in Figure 4.2. This instruction takes a vector register of doubles and an integer whose bits encode which elements in the vector should be compressed. It then stores these elements contiguously in memory.

We can emulate vcompresspd on CPUs with different instruction sets [28]. The compression can be used to extract the subsets from a single vector register. We can use this instruction twice to extract the elements of $S_1$ and $S_2$ from a vector register.

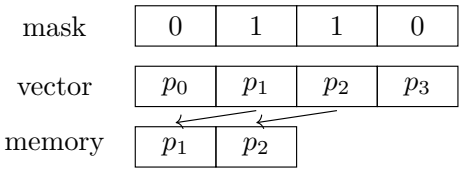



Figure 4.2: The vcompresspd instruction

To extend the extraction to an entire array, we repeatedly use the compression instruction while maintaining an invariant illustrated in Figure 4.3. This is Algorithm 10, which maintains—for $d$ the number of elements per register—that

1. we have $P[0, w_l) \subseteq S_1$,
2. we have $P[w_r, |P|) \subseteq S_2$,
3. either $r_l - w_l \geq d$ or $w_r - r_r \geq d$,
4. the unclassified points are in $P[r_l, r_r)$ or buffered.

**Algorithm 10** Subset extraction.

**Input:** A set of $n$ points $P \subset \mathbb{R}^2$.
**Output:** Pointers $w_l$, $w_r$ and a permutation of $P$ such that $P[0, w_l) = S_1$ and $P[w_r, |P|) = S_2$.

```
1:  w_l = 0; r_l = d; r_r = |P| − d; w_r = |P|;
2:  bufL = P[0, d); bufR = P[n − d, n);
3:  while r_l < r_r do
4:      if w_l − r_l ≥ r_r − w_r then
5:          reg = P[r_l, r_l + d)
6:          r_l = r_l + d
7:      else
8:          r_r = r_r − d
9:          reg = P[r_r, r_r + d)
10:     extract S_1 from reg, write to w_l, increase w_l
11:     extract S_2 from reg, decrease w_r, write to w_r
12: extract S_1, S_2 from bufL and bufR
```

We establish the invariant with step 1 and 2 of Algorithm 10. These buffered elements can be safely overwritten. Our writes respect assertions 1 and 2. We read $d$ elements from the side of the array where the distance between read and write pointer is smallest, ensuring that on that side no points are overwritten.

Assertion 3 ensures there are no points overwritten on the other side either, which implies 4. Therefore, we only have to show that assertion 3 holds. For this reason, we ensure at the initialisation of the algorithm that $(r_l - w_l) + (r_r - w_l) \geq 2d$. As we read $d$ elements at each step, and write less than or equal to $d$ elements, this inequality is maintained. The sum $(r_l - w_l) + (r_r - w_l)$ is at least $2d$, so the minimum $\min(r_l - w_l, r_r - w_l)$ is at least $d$.

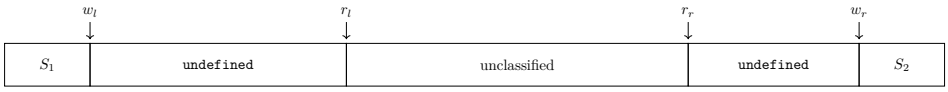



Figure 4.3: Invariant for extracting the subsets $S_1$ and $S_2$ from a set of points $P$.

## Parallel Extraction

Extracting $S_1$ and $S_2$ has linear complexity, so it is very likely to be limited by bandwidth. For this reason, minimizing data movement is the primary design goal of the parallel algorithm. The algorithm consists of two conceptual steps: a parallel step where each thread independently works on their local part of $P$, and a cleanup step that merges the subsets of each thread.

**Parallel step** We first divide the points $P$ over all threads. As we would like to have most points be in the correct position after the partitioning step, each thread should have points on the left and right side of the array. This makes the block-cyclic distribution, illustrated in Figure 4.4a suitable. To be more precise, the block-cyclic distribution over $T$ threads with block parameter $b$ partitions an index set $[0, n)$ into $T$ index sets

$$I^{(t)} = \{(pk + s)b + j \mid 0 \leq j < b, \quad 0 \leq (pk + s)b + j < n\}$$

Thread $t$ can then independently extract the subsets of $P^{(t)} := \{P_i \mid i \in I^{(t)}\}$. Figure 4.4b illustrates the resulting array and write pointers.

The key observation is that for

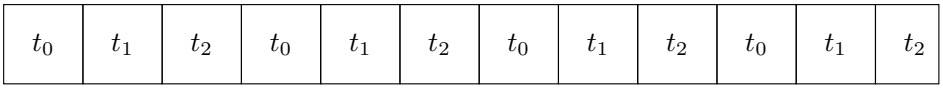
$$w_l^{\min} := \min_t w_l^t$$

$$w_l^{\max} := \max_t w_l^t$$

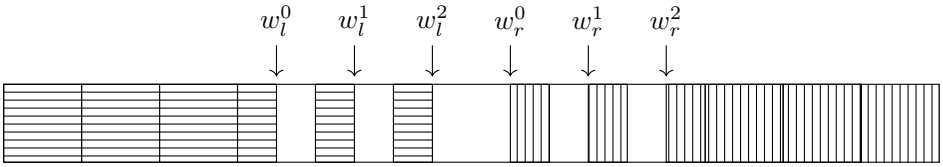
$$w_r^{\min} := \min_t w_r^t$$

$$w_r^{\max} := \max_t w_r^t$$

all points in $P[0, w_l^{\min})$ are in $S_1$, all points in $P[0, w_r^{\max})$ are in $S_2$, and all points in $P[w_l^{\max}, w_r^{\min})$ are undefined. This means only the points in $P[w_l^{\min}, w_l^{\max})$ and $P[w_r^{\min}, w_r^{\max})$ are potentially incorrect.




(a) Block-cyclic distribution. Each square represents blocksize $b$ elements, except the last one that may contain less.




(b) Result of the parallel step. The horizontal lines indicate $S_1$, the vertical lines $S_2$, and the blank space is undefined. The superscript indicates which thread the write pointers belong to.

Figure 4.4: Parallel partition step for 3 threads $t_0$, $t_1$, $t_2$

**Cleanup** For random input, or input that is already a convex hull, the write pointers of different threads will be close to each other. This means only few points are incorrect, so it is acceptable to classify the points in $P[w_l^{\min}, w_l^{\max}) \cup P[w_r^{\min}, w_r^{\max})$ sequentially. We use the Dutch National Flag algorithm for this problem. A challenge for this algorithm is that we need to test whether a point does not belong to $S_1$ or $S_2$. We overwrite points, so the actual values of those points is undefined. The only way to test this is by finding out to which thread the point belongs, and then checking whether the index is between the write pointers of that thread. To make this easier, each thread sets their undefined points to NaN.

## 4.4 Implementation

We implement the algorithm using the Highway library [28] for vectorisation, and OpenMP for multithreading. Our library is available with Fortran and C interfaces under a GPLv3 licence [146]. The basic idea is faithful to Algorithm 9 and the subset extraction, but several small changes are necessary to obtain good performance, which we document here.

**Parallelisation** We parallelize both the recursion and the subset extraction. We divide the available threads over the recursive calls in the same ratio as $|S_1| : |S_2|$. When we have only one thread left, we switch to the sequential implementation.

**Bandwidth considerations** To reduce the number of passes over the data, we compute the farthest points $r_1$, $r_2$ in the same pass as the subset extraction.

Writes to and from main memory are done in aligned chunks, typically 64 B, called cachelines. This has two consequences we need to mitigate. First, false sharing: two processors cannot write concurrently to the same cacheline, as one

processor would overwrite the result of the other. This causes an expensive cacheline invalidation. Second, partial writes to a cacheline still incur the full cost of writing to an entire cacheline.

To ensure each processor works on a disjoint cacheline, we pick the block size $b$ equal to a multiple of the cacheline. We also align $P$ to cacheline boundaries by processing a small number of elements at the start and end of the array that do not fill an entire cacheline sequentially.

CPUs try to mitigate partial transactions to main memory by accumulating writes to the same cacheline in a limited number of buffers. Writes to main memory are delayed as long as possible, so more data can be sent to main memory in a single transaction over the bus. On some of our test systems these buffers are overwhelmed because we write in an irregular pattern to four different memory regions ($x$- and $y$-coordinates from both the left and right side), and because we do many writes of irregular length that may cross cacheline boundaries. To give the CPU a better chance of combining bus traffic, we first write to small circular buffers that fit in the first-level data cache. These writes do not cause traffic to main memory. Once full, we can efficiently empty the buffers to main memory by using vectorized stores to aligned addresses.

## 4.5 Performance Evaluation

Comparing performance is only useful against the fastest available algorithm and implementation. The original authors of Quickhull, the Computational Geometry Algorithms Library [228], and the Problem Based Benchmark Suite [7] provide implementations of Quickhull, called Qhull, CGAL, PBBS respectively. CGAL also provides implementations for some of the other algorithms. Preliminary testing on a laptop (Table 4.1), shows that PBBS's implementation of Quickhull outperforms its closest competitor by a factor 2, so that is what we use as baseline.

| Implementation | Runtime |
|---|---|
| PBBS | 0.31 |
| CGAL Quickhull | 0.61 |
| CGAL Akl-Toussaint | 0.60 |
| CGAL Bykat | 0.73 |
| CGAL Eddy | 0.98 |
| Qhull | 1.1 |
| CGAL Graham | 1.3 |
| CGAL Jarvis | 208 |

Table 4.1: Runtime in seconds for a disk of $10^7$ points, sequential implementation.

The Problem Based Benchmark Suite also describes three datasets which we use for our evaluation. Each of these consists of $10^8$ points drawn from different distributions.

## Evaluation Platforms

We evaluate our implementation on three machines. The first, cn125, has powerful vector instructions, a modest amount of cores and memory, and a simple memory architecture. The second, cn132, has less powerful vector instructions and needs to implement the compression instruction in software. It has two CPUs which gives it more cores and memory bandwidth. One of these CPUs can access the other's memory, but at a performance penalty. This is called *non-uniform memory access* (NUMA). For this reason, we ensure allocations are equally distributed between the two CPUs by using numactl −−interleaved all. The third system, *laptop* has the same instruction set as cn132, but is a single consumer CPU. This CPU is a heterogeneous system with performance and efficiency cores, but we only use the former by running the experiments under numactl −C 0−7.

We summarize the systems in Table 4.2. There can be a significant gap between the bandwidth the RAM is capable off, and what is achievable in practice. The STREAM benchmark [168] is a collection of simple functions that can be used to measure what memory bandwidth is achievable in practice. To most accurately mirror the access pattern of VQhull, we report the Scale benchmark function, which reads and writes to the same array.

| | cn125 | cn132 | laptop |
|---|---|---|---|
| CPU | Xeon E-2378 | Epyc 7313 (×2) | i7-12700H |
| Cores | 8 | 16 (×2) | 8 |
| Vector extension | AVX-512 | AVX2 | AVX2 |
| Base–Boost frequency (GHz) | 2.6–4.8 | 3.0–3.7 | 2.3–4.7 |
| Theoretic bandwidth (GB/s) | 51.2 | 204.8 | 76.8 |
| STREAM (GB/s) | 41 | 140 | 52 |

Table 4.2: Theoretical and practical capabilities of test systems

## The Datasets

The performance of the Quickhull algorithm depends on the input distribution, so we briefly describe what makes the three datasets of PBBS interesting. This includes the structure of the recursion, and the number of bytes that must be moved. We compute the number of bytes as follows. Assuming a double is 8 bytes, this is $8n$ bytes for finding the left-most and right-most points. At each recursive call, we have $8(n + |S_1| + |S_2|)$ bytes for the partition, and $8|CH(S_2)|$ bytes to place $CH(S_1)$ and $CH(S_2)$ next to each other in memory. This is 5.6 GB, 73 GB, 7.2 GB for Kuzmin, Circle, and Disk respectively.

The first data set is drawn according to a Kuzmin distribution, which we illustrate in Figure 4.5. We only have one recursive call, and the recursion is very unbalanced.

The second data set consists of points sampled uniformly from a circle, so points of the form $(\cos(\theta), \sin(\theta))$. A true circle is convex, so all its points lie

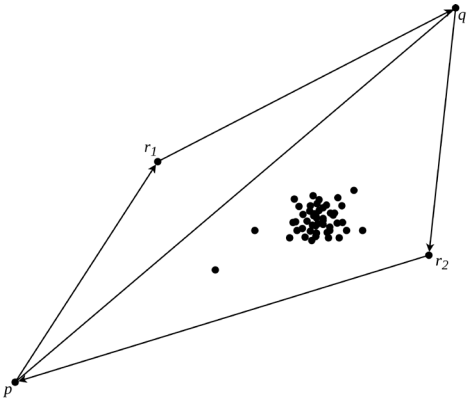



Figure 4.5: The Kuzmin dataset of $10^8$ points. The first partition only moves $r_1$. The second partition eliminates all remaining points.

on the convex hull. However, the algorithm finds only 10985400 points (∼11 %). This is not a mistake or a problem because a sampled circle locally resembles a line up to the square root of the unit roundoff, and points on a line can be rejected from the convex hull. To be more precise, at $(\cos(\theta), \sin(\theta))$, the tangent line has equation

$$\cos(\theta)x + \sin(\theta)y = 1$$

The distance between $(\cos(\theta + \delta), \sin(\theta + \delta))$ and this line is

$$|\cos(\theta)\cos(\theta + \delta) + \sin(\theta + \delta)\sin(\theta) - 1| = \cos(\delta) - 1 = O(\delta^2)$$

The unit round-off $u$ is $2^{-53}$ for double precision, so points on a circle that are approximately $\sqrt{u} \approx 10^{-8}$ apart, are indistinguishable from a line and even slight round-off errors create concave points. Regardless, the recursion is deep and balanced.

The last data set consists of points sampled uniformly on a disk, the interior of a circle. The recursion is also balanced, but because we discard more points the deeper recursion levels are less expensive compared to Circle.

## Metrics

Presenting performance results of an algorithm and its implementation in a meaningful way requires more than the runtime. The runtime alone tells us little about the quality of the implementation, as we cannot relate it to the capabilities of the machine. For this reason, we present our results using two statistics.

First—to evaluate how close the wallclock time gets to the hardware's peak—we compute the bandwidth, or the amount of data moved divided by time. We split the $x$- and $y$-coordinates into two separate arrays. Assuming 64-bit floating-point numbers are used, this is $8|P|$ bytes for finding the left-most and right-most points. At each recursive call, we have $8(|P| + |S_1| + |S_2|)$ bytes for the partition,

and $8|CH(S_2)|$ bytes to place $CH(S_1)$ and $CH(S_2)$ next to each other in memory. This results in 5.6 GB, 73 GB and 7.2 GB for Kuzmin, Circle, and Disk respectively. Second—we measure the energy consumption in Joules.

### Methodology

We use GCC 14.1.1. All experiments are run 10 times, and we plot the standard deviation with error bars.

In an attempt to generalize the energy consumption results across different machines, we subtract the 'idle' power draw of the system under test from the observed energy consumption. We define idle power draw as the baseline power that is required to keep the hardware operational, as well as the power required for the operating system and additional background tasks. After a five-minute measurement period, where both systems were fully reserved to ensure that no other tasks are running, the idle power draw of cn125 was found to be 3.7 W, and the idle power draw of cn132 was found to be 53.0 W for the first of the two CPUs, and 50.8 W for the second. The difference in power draw between these two nominally identical CPUs is expected, as silicon quality varies greatly even between CPUs of the same model [136]. *laptop* has an idle power draw of 5.4 W.

Intel and AMD both provide hardware counters that track the accumulated energy consumption of their processors. Although initially based on estimation models, these interfaces nowadays provide precise energy consumption statistics [134]. Intel CPUs provide these counters since their Sandy Bridge architecture (2011), whereas AMD started providing them with their Zen architecture (2017). These counters track the energy consumption of the entire CPU, including the uncore and DRAM controller. Although ideally we would also measure the energy consumption of the memory bus and the DRAM itself, these capabilities are not available to us.

## Differences with PBBS

The PBBS implementation partitions an array of indices, instead of the points themselves. Their parallelisation strategy is to spawn threads on the recursive call until a base case of 400 points is reached. The points $r_1$, $r_2$ are found in a separate pass.

The partition algorithm applies Hoare's algorithm [110] sequentially, and is scan-based [32] in parallel. Because of this, runtime speedup and energy-efficiency improvements of PBBS are much greater when comparing the parallel case against the sequential case than for VQhull, which applies the same approach in both the sequential and parallel case.

Finally, VQhull uses separate arrays for $x$- and $y$-coordinates, whereas PBBS uses a single array of data type:

```
struct {
    double x;
    double y;
};
```

## Computational Performance Analysis

**Branching** For the Kuzmin dataset almost all points belong in either $S_1$ or in $S_2$, so the branches of Hoare's partition algorithm are easy to predict. VQhull on the other hand only has one difficult branch every 4 or 8 elements. This is reflected in Figure 4.7, where the sequential PBBS implementations differs as much as a factor 6 between different inputs, while VQhull differs by at most 20 %.

**Vectorisation** Even if branch misprediction and memory bandwidth are not an issue, so for sequential Kuzmin, the VQhull implementation is still 1.5 ×–3.2 × faster than the PBBS implementation, as SIMD instructions can compute the tests 4 or 8 points at a time. We do not get a speedup this large as the compression instruction is not 4 ×–8 × faster than the non-vectorized counterpart, and because some SIMD instructions can be used within a single test as well.

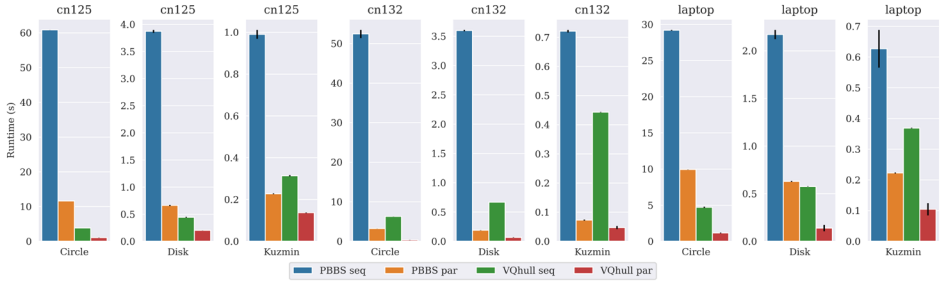



Figure 4.6: VQhull and PBBS runtime in seconds.

## Memory Subsystem Analysis

The weaker parallel performance of PBBS can be explained by the actual amount of traffic over the bus being higher than the minimum amount we use for our calculations. Finding $r_1$ and $r_2$ each take an additional extra pass over the data, which explains why VQhull is faster. As the points are not permuted, only part of the cacheline is used effectively at deeper recursion levels of Disk and Circle. This explains why they get lower bandwidth than Kuzmin for PBBS.

For VQhull we obtain 98 %–100 % of STREAM's bandwidth for Kuzmin on non-NUMA systems, and 85 %–90 % for Disk. Disk's performance may be lower because it is unpredictable from which side of the array we read, making prefetching less effective. For Circle, we exceed the RAM's bandwidth because significant work is done on subsets fitting in cache.

On cn132, a dual socket system with a NUMA architecture, we only obtain 66 %–80 % of STREAM. Profiling shows the first subset extraction is done at 100 % of STREAM, but deeper recursions are significantly slower. This can be explained by suboptimal locality of threads spawned in recursive calls, which we do not control.

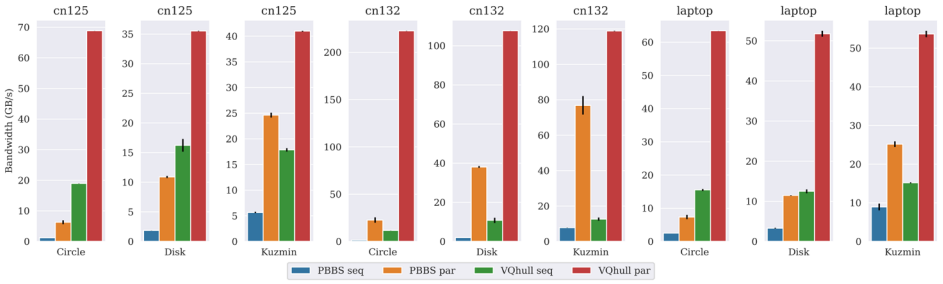



Figure 4.7: VQhull and PBBS bandwidth in GB/s.

## Energy Consumption Analysis

Usually, going from the sequential implementation to the corresponding parallel implementation decreases the total energy consumption. At first sight this seems counter-intuitive, as in the end the same amount of work is being done by both implementations, albeit with the work being distributed over multiple threads in the parallel case. If anything, one might expect the additional energy overhead related to scheduling to result in an increase in energy consumption. However, CPUs typically run at a higher frequency for single-threaded workloads than for parallel workloads. Because power draw is roughly proportional to cubic frequency [135], this explains why parallelizing a workload, and consequently computing at a lower frequency, can decrease total energy consumption.

The only two exceptions to this observation occur on cn125, for the Disk and Kuzmin datasets. For these two cases, even though the runtime decreases going from sequential VQhull to the parallel implementation, the energy consumption slightly increases. This highlights a case where an increase in energy consumption, due to increased complexity of the implementation, as well as additional operating system overhead, is not overcome by hardware efficiency improvements resulting from a parallelized approach.

Furthermore, we see that even within a single implementation, a decrease in runtime does not correlate to a decrease in energy consumption. Although sequential PBBS is faster on cn132 than on cn125 for every dataset, it conversely consumes more energy. Compared to *laptop*, this is only the case for the Kuzmin dataset. Whereas for Circle and Disk, *laptop* is both faster and more energy-efficient. This highlights that observing runtime performance alone does not give the whole picture. And that selecting the optimal implementation of an algorithm is highly context-dependent, depending not only on the implementation itself but also on the hardware.

Finally, our experiments show that optimizing an implementation has gains beyond improving runtime performance. Although parallel PBBS on average has a shorter runtime than VQhull, in all but one case sequential VQhull results in significantly lower energy consumption. This observation shows that decreasing the overall amount of branch-misses and computational work, as well as making more effective use of the available hardware and instruction set, allows for substantial

gains in energy-efficiency, even when runtime speedups are minimal.

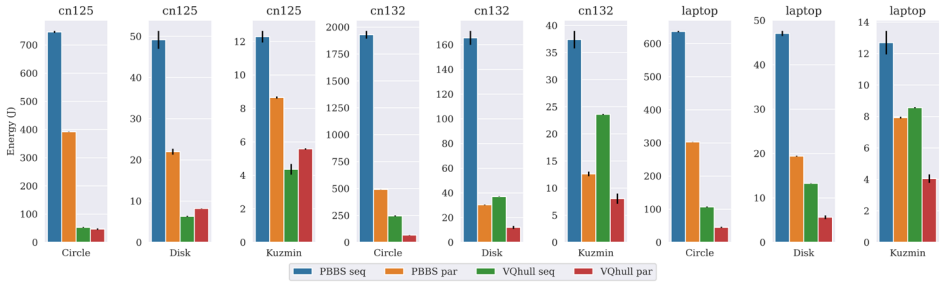



Figure 4.8: VQhull and PBBS energy consumption in Joules.

## 4.6 Related and Future Work

Finding planar convex hulls is a well-studied problem, there exist at least ten algorithms for solving it [92, 125, 70, 196, 42, 4, 10, 53, 37, 48]. Putting our work in context also suggests several opportunities for future work.

The sequential subset extraction is a straightforward adaptation of the vectorized partition algorithm used in QuickSort [38].

Our analysis shows the implementation VQhull is limited by bandwidth, and that it gets close to the theoretic maximum any implementation of Quickhull can obtain. For this reason, significant speedups can only be obtained by implementing an algorithm with less data movement than Quickhull. With this in mind, the Akl-Toussaint heuristic [4] would be an interesting opportunity for future work. In the case of Disk, this heuristic lets us identify $p_1, \cdots, p_8$ of Figure 4.9 in one pass. This octagon has area $2\sqrt{2}$, which is approximately 90 % of the disk, and these points can be eliminated. Points $u$ satisfy $\text{orient}(p_i, u, p_{i+1 \bmod 8}) > 0$ for precisely one $i$ if they are between the circle and line segment $\overrightarrow{p_i p_{i+1 \bmod 8}}$, or none if $u$ is inside the octagon. If we can partition the input into these eight areas in one pass as well, we need only $1.6 + 1.6 + 0.16 = 3.36$ GB to eliminate 90 % of points. VQhull gets to the same points in 3 levels of recursion, which takes 6.5 GB. So a better algorithm can cut the required bandwidth almost in half.

There is some work on computing convex hulls on Graphic Processor Units (GPUs) [224]. They report a 16× speedup compared to Qhull, which is not competitive with VQhull. However, single-threaded performance has not increased nearly at the rate of GPU performance, so this may have changed. Their implementation is not publicly available.

## 4.7 Conclusion

We present VQhull, a parallel algorithm for Quickhull Our implementation shows an improvement of up to 16 × compared to the state-of-the-art sequential im-

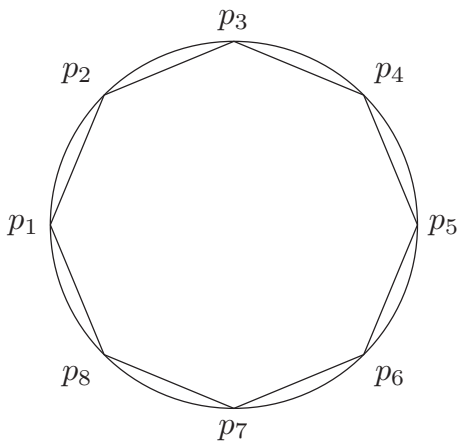



Figure 4.9: Points $p_1, \cdots, p_8$ can be found by minimum / maximum $x$-coordinates, $y$-coordinates, differences / sums of $x$ and $y$ coordinates. Points within this octagon are not in the convex hull.

plementation. By focussing on the memory subsystem instead of computational parallelism, also the multithreaded implementation shows an improvement of up to 11 × improvement over the state-of-the-art parallel implementation.

Our implementations achieve 66 %–100 % of the architecture's peak bandwidth, showing the potential for further improvements is 0 %–50 %. VQhull does best on systems with a single CPU, and an instruction set that supports a compression instruction. A system with multiple CPUs and a non-uniform memory architecture shows significant performance degradation for nested OpenMP parallelism, providing an opportunity for future work.

This algorithm also serves as a case study on engineering algorithms for energy-efficiency. We show that parallelizing programs is beneficial to energy-efficiency, resulting from a decrease in clock frequency. Energy savings on specific architectures are mostly in line with runtime improvements, but even when runtime savings are minimal, a well-thought-out implementation can still result in significant energy savings.

# Subpart B

# Code Generation

# Chapter 5

# Shray: An Owner-Compute Distributed Shared-Memory System

## 5.1 Introduction

Orchestrating parallel programs for clusters of computers is known to be a challenging task. This is despite clusters being relatively simple on a hardware level: we essentially have a collection of computers that can send data between two computers. The difficulty is that the hardware does not reflect our computations very well.

Many approaches towards closing this gap have been developed. These approaches range from programs mirroring the hardware closely, making all data placement and all communication explicit, to approaches that hide the hardware behind a virtual shared-memory machine, where all decisions about data placement and communication are hidden from the programmer. MPI (Message Passing Interface) is one of the best known examples of the former, whereas Distributed Shared Memory Systems (DSMs) aim at the latter.

In practice, a hybrid approach named PGAS (for partitioned global address spaces) [63] has gained a lot of traction as it seems to strike a good balance between programmability of DSM systems, and efficiency of explicit approaches. PGAS introduces the notion of ownership of partitions of data by individual compute nodes. Reads and writes to locally owned memory can be done without any communication whereas reads and writes to remote parts require communication which is issued and orchestrated by compilers and run-time systems.

Within the many PGAS-based systems, we see a wide range on how explicit the notion of the data distribution is. Some systems require the programmer to explicitly distinguish between local and non-local reads and writes, other systems allow users to shift this distinction into the compilation process or even the run-time system. In particular the latter typically gives rise to rather sophisticated

implementations (typically tens of thousands of lines of code) in order to achieve good levels of parallel performance.

In this chapter, we propose a different point in the design space. It is based on the following observation: in scientific computing there are many parts of algorithms that we can make efficient without intricate control over the hardware. For example point-wise operations and low-dimensional stencils are easy to adapt to a distributed system. For these parts of the algorithm, a DSM-like approach will give us easy to maintain and debug code, at no performance cost. For other algorithms, it is vital to have exact control over communication and synchronisation, which we do not have in DSM or PGAS systems.

Instead of picking one level of abstraction, we embed a high-level DSM system inside a low-level system. The programmer can then choose for each data-structure what approach is most appropriate. No longer needing to support all programming patterns, we consider possible restrictions, provided they allow for simplicity in the specification of most parallel algorithms, while enabling a leaner and possibly more performant implementation of the runtime system. To this end, we propose a new form of DSM, named *owner-compute DSM* or oc-DSM for short. Similar to PGAS, the key idea is to introduce a notion of ownership of data. However, there is no need to consider ownership when it comes to reading data. On reads, oc-DSM behaves like pure DSM systems. When it comes to writes, we impose a strict restriction: each node can only write to data it owns. Write operations anywhere else lead to undetermined behaviour.

The motivation of this approach is mainly based on three observations: Firstly, the restriction to owner-compute enables consistency of our DSM through synchronisations only, without requiring any sophisticated data exchanges between nodes. This alleviates many performance and scalability problems of classical DSM systems. Secondly, many parallel applications are or at least can be structured around systematic computations of new values while read accesses tend to be more irregular. This should render the owner-compute rule a minor restriction, be it for a manual use of oc-DSM or for a use as target of code generators from higher-level languages. Finally, uniform global reads can be achieved by leveraging the hardware-capability of the memory management unit (MMU). Using MMU features jointly with a communication layer such as GASNet allows for an implementation of the locality-check on data as well as all necessary communication without any user code other than a normal selection at all. We allow this oc-DSM to operate on a per-array basis, and integrate it with our explicit host system.

This chapter makes the following contributions:

- It introduces the oc-DSM model that is simple to use and understand, both by a programmer and by code generators.

- It sketches an implementation of oc-DSM in the form of a C library named Shray (for **Sh**ared Ar**ray**s) of as little as 800 lines of code. It builds on the mmap system interface for controlling the MMU behaviour and on GASNet as communication conduit.

- We demonstrate how the oc-DSM model enables the seamless use of many external libraries and tools, even if these are totally unaware of the distributed memory context they are to be used in. This includes the use of OpenMP for generating an additional level of local parallelisations.
- By implementing Shray as a library, it can be used alongside the host system for parts of the application not suitable for array programming.
- Finally, we provide an initial performance evaluation, comparing Shray-performance with several state of the art PGAS tools.

In Section 5.2 we describe the core design principles of oc-DSM. Then, in Section 5.3 we explain some key ideas behind our implementation Shray.

In Section 5.4 we show that in several example applications parallel formulations in C using the Shray-interface are surprisingly close to their sequential counterparts when comparing them with their counterparts in PGAS languages such as Global Arrays [185], Chapel [43] and UPC [86].

Finally, we also present some performance comparisons on a small cluster in Section 5.5, looking at some initial indications of the performance potential of the proposed approach. They show that a level of performance can be achieved, that is competitive and in some cases even superior to that of the PGAS counterparts.

## 5.2 Design Principles of Oc-DSM

64-bit architectures can operate on a virtual address space of $2^{64} = 16$ Exa-Byte which even today is more than the total amount of memory available in the largest supercomputers ($9.2 * 2^{50}$ bytes for Frontier [188]). [1] Although individual machines typically have much less physical memory attached to them, the MMUs of cluster nodes enables operating systems to freely map memory regions of the huge virtual address space to the smaller physically available memory. This hardware capability enables allocations of virtual memory without the necessity to map all of it to physical memory. If the software tries to access non-physically-allocated portions, an exception (SEGFAULT) is being raised by the MMU which subsequently can be dealt with in software.

Like in classical DSMs, one key idea of oc-DSM is to leverage this readily available mechanism for a simple yet effective implementation: we allocate shared data-structures on *all* participating cluster nodes in their *full* size in virtual memory. Building on the idea of partitioned data in an SPMD-style program, we divide the total allocated memory up into as many blocks as we have participating cluster nodes. Each of the nodes then attaches physical memory to the block of memory that it "owns".

Since we allocate distributed data fully in virtual memory, from the software perspective, all elements, including those that are located in physically remote memory, can be accessed by any of the CPUs by performing a normal read operation. If a read into a locally not available address range occurs, a SEGFAULT

[1] Intel's X-86 architecture only supports $2^{48}$ bytes which still equates 256 TB.

is issued by the MMU which can be used to fetch remote data, and put it into physical memory that is attached to the corresponding portion of virtual memory. Any subsequent reads into such fetched-from-remote data will come without any further overhead.

A second key design decision of oc-DSM is the owner-compute rule. We demand all write operations to be restricted to the local portion of any distributed data structure. That way, we can obtain consistency without the need to exchange any data between nodes. All we need to do is to invalidate any possibly fetched remote data that has been updated in a write-phase to a shared data structure. We can identify the end of such write-phases through explicit synchronisations.

These two design principles allow for rather straight-forward implementations; mainly providing a new version of malloc for allocating distributed data structures and a signal-handler that handles SEGFAULTS whenever a program tries to access remote data. This DSM implementation is done purely in user-space software, and no explicit communication needs to be inserted in executed code. This is especially useful when working with external libraries or code generators.

The key challenges in such an implementation are memory scalability and multi-threading. Memory scalability here relates to the observation that when the number of participating nodes increases, we risk running out of physical memory on individual nodes when reading from too many other nodes. Consequently, we need to handle physical memory like a cache and put some eviction policy in place which frees up memory that is not "owned" and deemed to be no longer needed.

Multi-threading also requires additional attention as this can lead to several concurrent reads into the same non-local memory. In such situations, we need to avoid multiple loads of the same pages and we also need to make sure we never read from pages that are not yet fully initialised.

## 5.3 Shray, an Implementation of Oc-DSM

We have created a free open-source implementation of oc-DSM named Shray [216], which stands for **Sh**ared Ar**ray**s, emphasising its primary intended use for large distributed array-like data-structures.

### The Programming Interface of Shray

The key challenge in the design of the application programming interface of Shray is to find a way to create an abstraction that makes it easy for programmers to adhere to the owner-compute restriction without being required to perform major program refactoring when trying to parallelise an existing sequential algorithm. We use a trivial example to expose our key idea: In Listing 5.1 we show a C program that allocates an array and then adds one to every value. For the sake of the example, we assume that the array is two dimensional and that the actual value-modifying code in arr_inc is structured accordingly, i.e., as two nested for-loops.

If we were to parallelise such a code manually, we would aim to schedule the computation by distributing the outer loop across the nodes as this creates good spatial locality and coarse granularity of tasks. This observation shows that we can easily adhere to the owner-compute rule if we make sure that our partitioning does have a granularity of multiples of the size of our rows, n elements in the example. If we ensure the partitioning happens between rows, this code can be parallelised by restricting the outer for-loop to those rows that are local to any given node, and by running the entire code in SPMD fashion.

```
void arr_inc(size_t m, size_t n, double (*x)[n]) {
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n; j++) {
            x[i][j]++;
        }
    }
}

int main(int argc, char **argv) {
  double (*x)[n] = malloc(m * n * sizeof(double));
  ... arr_inc(m, n, x); ...
  free(x);
}
```

Listing 5.1: Sequentially incrementing a 2d array

This simple insight guides the design of the interface of Shray, whose key functions are shown in Listing 5.2.

```
/* initialise Shray */
void ShrayInit(int *argc, char ***argv)

/* allocate a distributed array */
void *ShrayMalloc(size_t firstDimension, size_t totalSize)

/* deallocate a distributed array */
void ShrayFree(void *array);

/* request first owned index */
size_t ShrayStart(void *array)

/* request upper bound (exclusive)
 * for owned indices */
size_t ShrayEnd(void *array)

/* perform a global synchronisation */
void ShraySync(void *array, ...)

/* finalise the use of Shray */
```

```
void ShrayFinalize(int exit_code)
```

Listing 5.2: Shray API

We can see that besides some initialisation and finalisation code in ShrayInit and ShrayFinalize, the main interface functions are a shared allocation and deallocation ShrayMalloc and ShrayFree, as well as the three functions ShrayStart, ShrayEnd, and ShraySync. To accommodate our need to express a minimum granularity, we provide an additional parameter firstDimension to ShrayMalloc. It specifies how many chunks the array is to be distributed over. For our running example, we can simply use the number of rows m.

Once a distributed array has been allocated, we can query Shray as to which rows (or hyperplanes for
higher-dimensional arrays) are owned by the current node. This happens through the functions ShrayStart and
ShrayEnd. Finally, we have to synchronise whenever we finish writing into an array to make sure that we establish consistency for the given array. Applying this technique to our running example leads to the code in Listing 5.3.

```
void arr_inc(size_t m, size_t n, double (*x)[n]) {
    for (size_t i = ShrayStart(x); i < ShrayEnd(x); i++) {
        for (size_t j = 0; j < n; j++) {
            x[i][j]++;
        }
    }
    ShraySync(x);
}

int main(int argc, char **argv) {
  ShrayInit(&argc, &argv);
  double (*x)[n] = ShrayMalloc(m, m * n * sizeof(double));
  ... arr_inc(m, n, x); ...
  ShrayFree(x);
  ShrayFinalize(0);
}
```

Listing 5.3: Incrementing a 2d array with Shray

Note here that the synchronisation is specific to the array in question. It formally guarantees that all nodes have finished writing their share of that array. That way read-operations into remotely located parts of a shared array can be guaranteed to return the values written before the last ShraySync, and write operations can be guaranteed not to overwrite data that is still needed by some remote node.

Another aspect to notice here is that we do not make any assumptions about the number of participating nodes. All this is hidden inside of Shray allowing for runs on different node configurations without any recompilation or code adaptation. The use of ShrayStart and ShrayEnd implicitly takes this aspect into account.

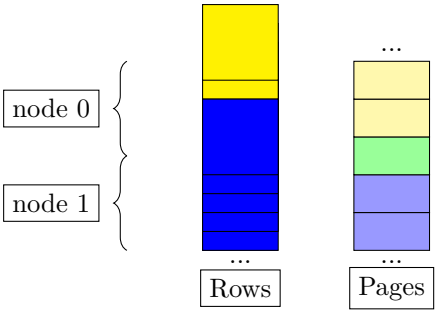



Figure 5.1: Distribution of an array of doubles over two nodes, depicted in yellow and blue. On the left-hand side we show the rows, on the right-hand side the pages. As the middle page contains rows from both nodes, it is shared.

## The Implementation of Shray

MMUs handle memory in indivisible chunks of memory referred-to as pages. These are typically 4 KiB large but can differ from system to system. Any memory protection and consequently any SEGFAULT can only be issued at that granularity. This is at odds with the liberty of Shray to declare arbitrarily sized granularities for the address space partitioning, which potentially may not be a divisor of a given page size. To deal with this issue, Shray under the hood has a notion of pages that have a shared ownership. Such shared pages can only occur at the boundary between two (or more) nodes. We visualize this in Figure 5.1. Both nodes own part of the middle page, so they will not SEGFAULT even on parts of the page they do not own.

If a distributed array has co-owned pages then any synchronisation through ShraySync requires an exchange of updated data between the two nodes that own such a shared page. Fortunately, this requires only a maximum of two one-page communications for each node, irrespective of the overall size of the cluster. It also requires an invalidation of non-owned memory, but this is practically a constant-time operation.

Dealing with memory scalability is needed to ensure that individual nodes do not run out of physical memory beyond the operating systems ability to perform swapping or, preferably, to not engage with swapping at all. In Shray, we enable the programmer to restrict the amount of physical memory that Shray is allowed to allocate per shared array on a given node. This limit enforces Shray to discard memory that has been fetched earlier when too many remote reads occur. The data structures necessary to keep track of this are presented with a concrete example in Figure 5.2. We use two core data structures per distributed array. We keep a bitmap PageFetched where each bit keeps track of whether we have a page locally or not. This is necessary to check whether another thread is already fetching a page. At first glance this may not seem memory-scalable, but we abuse the fact that mmap is a lazy allocator to make sure unset bits are not actually stored. To keep track of the order in which pages have been fetched,

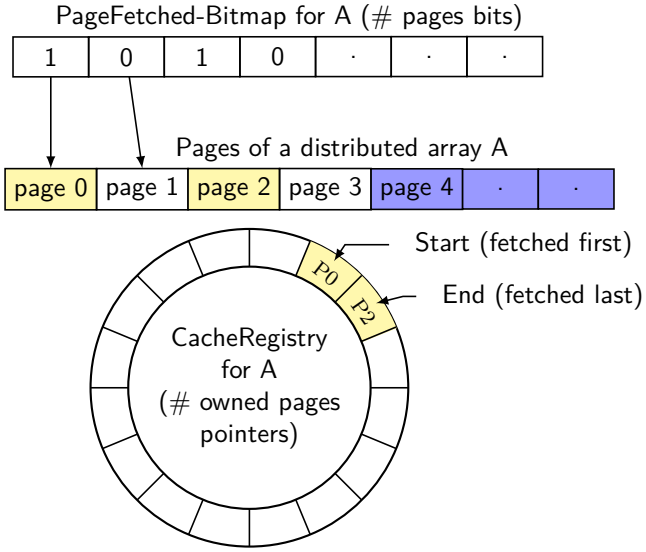



Figure 5.2: Data structures to track information about a distributed array, from the perspective of the blue node where page 0 and page 2 have been fetched from the yellow node and stored in the cache.

we have a FIFO queue implemented as a ringbuffer. The current implementation discards individual memory pages in a FIFO manner, effectively implementing a cache. Consequently, the size of the ringbuffer controls the total amount of physical memory that can be used for any given array. We choose this size to be a multiple of the locally owned memory portion and allow the user to control this factor through an environment variable named SHRAY_CACHEFACTOR which defaults to one.

Depending on the access pattern this can lead to higher-than-expected access times to remote data that has been accessed earlier but whose page has been discarded in the meantime. Fortunately, this form of run-time penalty for lacking spatial and temporal locality is well-known from the behaviour of hardware caches. Consequently, many algorithms employ techniques such as blocking to improve locality.

Likewise, many compiler optimisation techniques such as loop interchange [5], loop skewing [244], data reordering, etc. improve the locality of memory accesses further. All these techniques immediately benefit the effectiveness of Shray as well.

An unexpected challenge of this approach is that most mmap implementations have an upper limit on the number of virtual memory mappings. To avoid reaching this limit, for example by fragmentation due to multiple non-consecutive mappings, we introduce a customisable internal notion of page size within Shray, which can be set to a fixed multiple of the system's page size. This can be configured through an environment variable SHRAY_CACHELINE.

## Multithreading Interoperability

Shray is fully thread-safe and can be combined with OpenMP and pthreads as long as the owner-computes principle is observed. In the context of virtual memory and the MMU, thread-safety poses some challenges. There are two main issues, namely ensuring that we do not read garbage values due to modified page permissions and ensuring we do not fetch the same page multiple times.

For the former, when a SEGFAULT occurs Shray must mark the specific page as writeable before being able to copy remote data. However, on some architectures like x86 a writeable page is automatically readable as well. This introduces a race condition where thread one triggers a SEFGFAULT and while copying the data, thread two reads from the same page. To avoid this problem, we copy remote data to a shadow page first, leaving the original page attributes intact. Once copying is finished we then remap from the shadow array to the original address.

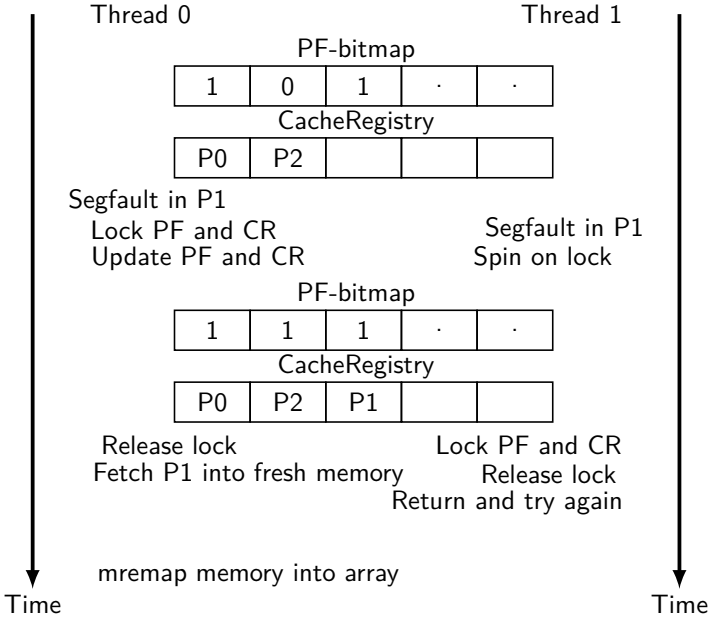



Figure 5.3: Handling of SEGFAULT in a multi-threaded context. Both threads access an address in page 1 which has not been fetched. Thread 0 triggers the fetch, Thread 1 tries again until the data is present.

To avoid multiple remote fetches of the same page, we use the PageFetched structure which, jointly with the cache registry, is manipulated within a critical section. As shown in Figure 5.3, within this section we check if we have already fetched the page, and if not, we trigger the remote load using a shadow page. We use a spinlock to guard the critical section. Other types of locks, such as a standard pthread mutex, can not be used as they are not safe to use inside a signal handler.

## 5.4 Programmability Comparison

We compare the ease of implementation of Shray with other PGAS languages by looking at possible ways to parallelise matrix multiplication.

One of the key driving factors in the design of oc-DSM is to use normal selections for all reads, local and remote. This does not only help in developing parallel versions of existing sequential implementations, but it also enables a direct utilisation of pre-existing libraries. As an example, consider a matrix multiply implemented through the BLAS [30] routine DGEMM as shown in Listing 5.4. The function multiplies two square matrices of size $n \times n$ and stores the result in $C$.

The first three arguments $n$ to the call of cblas_dgemm define the number of rows of A and C, the number of rows of B (which is the number of columns of A) and the number of columns in C. The other occurrences of $n$ as argument denote the row lengths of the array arguments which they follow.

```
void matmul(double *A, double *B, double *C, size_t n) {
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
              n,
              n, n, 1.0, A, n, B, n, 0.0,
              C, n);
}
```

Listing 5.4: Wrapper around DGEMM

All that is needed to run this in parallel using Shray, is to allocate the arrays through the Shray-layer using the row-length $n$ as granularity, and to change the arguments to cblas_dgemm. Each node calculates the local part of C using the local part of A only. This means we have to adjust the addresses of A and C as well as the first parameter $n$ that we pass in, resulting in the code shown in Listing 5.5.

```
void matmul(double *A, double *B, double *C, size_t n) {
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
              ShrayEnd(A) - ShrayStart(A),
              n, n, 1.0, A + (ShrayStart(A) * n), n, B,
              n, 0.0, C + (ShrayStart(C) * n), n);
}
```

Listing 5.5: Parallelising DGEMM with Shray

As we can see, we can conveniently obtain the start and end indices from Shray using calls to the functions ShrayStart and ShrayEnd, respectively. Some basic address arithmetic then enables us to restrict the computation to the local section of C.

Performing the same modification is not as trivial in other languages and libraries such as UPC, Chapel and Global Arrays. None can call DGEMM directly on distributed arrays, so we need to copy out the arrays to non-distributed memory. Doing this naively would necessitate creating a local buffer the size of $B$

and explicitly fetch the remote data. Such a solution is not memory scalable. To avoid this problem we must undergo a non-trivial modification of the algorithm. One possible approach is to use a blocked algorithm based on the following way of writing matrix multiplication.

$$\begin{pmatrix} A_{0,0} & \cdots & A_{0,p-1} \\ \vdots & \ddots & \vdots \\ A_{0,p-1} & \cdots & A_{p-1,p-1} \end{pmatrix} \begin{pmatrix} B_0 \\ \vdots \\ B_{p-1} \end{pmatrix} = \begin{pmatrix} \sum_{l=0}^{p-1} A_{0,l} B_l \\ \vdots \\ \sum_{l=0}^{p-1} A_{p-1,l} B_l \end{pmatrix}$$

We evaluate the sum $\sum_{l=0}^{p-1} A_{s,l} B_l$ one term at a time, copying out only $B_l$ from shared memory to local memory. The UPC implementation of this is given in Listing 5.6.

First we allocate a buffer large enough to hold $B_l$. We then introduce a loop over all processing elements where in each iteration we fetch $B_l$ and store it into the local buffer. With this we can call DGEMM on the submatrix.

Chapel is a higher-level language, having constructs such as localSubdomain in order to specify local parts of distributed arrays. It also does not use the SPMD paradigm, meaning we need to explicitly specify what to do in a distributed manner with coforall (but no if (myRank == 0) type construct common in SPMD programs).

```
void matmul(double *A, shared double *B,
            double *C, size_t n) {
  int p = THREADS;
  double *Bl = malloc(n / p * n * sizeof(double));
  for (int l = 0; l < p; t++) {
    upc_memget(Bl, &B[l], n / p * n * sizeof(double));
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                n / p, n, n / p, 1.0,
                &A[l * n / p], n, Bl, n, 1.0,
                C, n);
  }
  free(Bl);
}
```

Listing 5.6: Parallelising DGEMM with UPC

```
const Space = {1..n, 1..n};
const BlockSpace = Space dmapped Block(boundingBox=Space,
                                       targetLocales=MyLocales);
var A: [BlockSpace] real = 1;
var B: [BlockSpace] real = 1;
var C: [BlockSpace] real;
coforall loc in Locales do on loc {
  for l in 0..numLocales - 1 {
    var As: [1..n / numLocales, 1..n / numLocales] real =
      A[A.localSubdomain()](..,
```

```
            1 + l * n / numLocales..(l + 1) * n / numLocales);
        var Bl: [1..n / numLocales,1..n] real =
            B[B.localSubdomain(Locales[l])];
        C[C.localSubdomain()] += dot(As, Bl);
    }
}
```

Listing 5.7: Parallelising DGEMM with Chapel

## Distributions and Array Layouts

All our examples so far make two implicit assumptions: higher-dimensional arrays are stored row-major in memory and we do have a block distribution on the first axis.

However, a block distribution over the first axis is not always desirable. To illustrate this, let us consider two ways of distributing a 2d array over four nodes. The left-hand side of Figure 5.4 shows a block distribution over the first axis, while the right-hand side shows a block distribution over both axes.

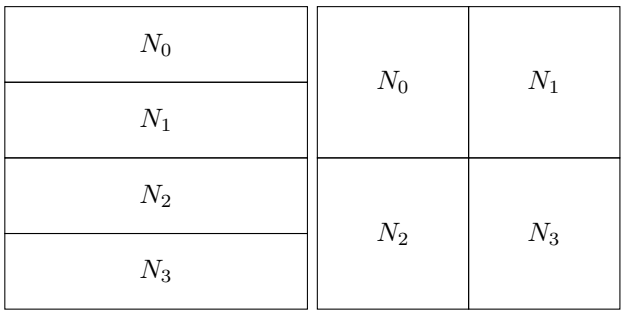



Figure 5.4: Two ways to distribute a 2d array over four nodes $N_0, N_1, N_2, N_3$. On the left-hand side a block distribution over the rows, on the right-hand side a 2d distribution over both rows and columns.

Though the left-hand side distribution is fine for some applications, others prefer the right-hand side distribution. One example of this is matrix-multiplication.

Suppose we want to compute $C = AB$ where $A$, $B$, $C$ are $n \times n$ matrices on $q^2$ nodes. In order to compute a row of $C$, we need all of $B$. So the communication is $O(n^2)$ for each node regardless of how many nodes we use.

If we compute a $\frac{n}{q} \times \frac{n}{q}$ block however, we only need 1 in every $q$ rows and 1 in every $q$ columns, which gives asymptotically superior $O(\frac{n^2}{q})$ communication.

If we wanted to extend Shray to support such distributions natively, we would not only have to extend the interface of Shray significantly, but we would also face a technical problem: The owned part of an array is a memory region with different protections from the non-owned parts. The kernel does some book-keeping for each contiguous block of memory with the same protections. That means that a

scalable implementation of oc-DSM must ensure that the owned part of an array is one (or a small fixed number) of contiguous blocks of memory.

While this observation might seem to be rather limiting, it actually is not. The key insight is that we can achieve arbitrary distributions as long as we change the data layout accordingly, i.e., we need to make sure that the individual blocks are located in consecutive address ranges. In our example from above, this means that we need to use a blocked data layout as indicated by the curve of Figure 5.5. Note here, that this is similar to the blocked layouts that are being used in optimised CPU versions [91].

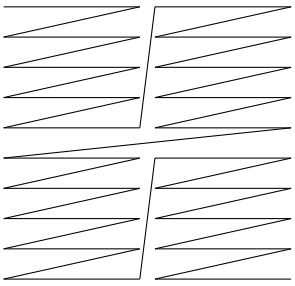


Figure 5.5: The curve represents the flattening in memory. Each of the four submatrices are stored contiguously.

A helpful way of thinking about this layout, is to see the array as conceptually three-dimensional. The shape would become $[q^2, \frac{n}{q}, \frac{n}{q}]$, where x[t, i, j] would index the $(i, j)$th element of node $t$'s submatrix. The allocation would be done with ShrayMalloc(q $*$ q, n $*$ n $*$ sizeof(double)). If two-dimensional labelling of the nodes, (so $N_{0,0}$, $N_{0,1}$, $N_{1,0}$, $N_{1,1}$) is preferred, one can split up the loop of $q^2$ iterations into two nested loops of $q$ iterations each by writing

```
for (size_t i = ShrayStart(x); i < ShrayEnd(x); i++)
{
    size_t i0 = i / q;
    size_t i1 = i % q;
    ...
}
```

and then using i0, i1 as loop variables.

If for some data structures a technique like this is not so straightforward, or desirable for performance reasons, the programmer can choose to trade off programmability for performance, and implement that part of the program in the host system.

## 5.5 Performance Evaluation

To evaluate the performance of Shray we look two types of benchmarks. The first type, consisting of micro-benchmarks, provides insights into the overhead of

Table 5.1: Versions of the tested software

| Language/Library | version |
|---|---|
| GASNet | 2023.3.0 |
| Global Arrays | 5.8.2 on MPICH |
| UPC | 2022.10.0 |
| Chapel | 1.31.0 |
| OpenBLAS | 0.3.20 |
| OpenMPI | 4.1.2 |
| MPICH | 4.0 |

our DSM layer. For the second type we look at some algorithms in numerical computing and compare these to alternative solutions. Performance of parallel systems can differ significantly between algorithms, so we have chosen the following two for initial performance measurements.

- Dense Matrix Multiplication (DGEMM) [161]: An algorithm requiring significant communication.
- Conjugate Gradient [16]: An algorithm with an unpredictable access pattern.

We compare Shray with implementations in UPC, Global Arrays, and Chapel. These languages and libraries are commonly used, are based on a lot of performance tuning research, and provide different approaches compared to Shray. Global Arrays is similar to Shray in that it is a library but it uses PGAS instead of a DSM. UPC on the other hand is an extension to C and has its own compiler. Chapel represents a high-level language approach with a completely different language and compiler.

## Setup

We perform experiments on a small cluster of 8 nodes. Each node contains an 8-core Intel Xeon E-2378 running at 2.60 GHz and 16 GB of RAM. They are connected via a single 1 Gbit/s NetXtreme BCM5720 Gigabit Ethernet PCIe interconnect. All programs that use BLAS routines use the OpenBLAS [242] library under the hood. Our cluster does not have RDMA hardware, so we use the MPI conduit with GASNet. A full overview of the software versions is given in Table 5.1.

For the DGEMM and Conjugate Gradient benchmarks we maximise the resources of a single node first. This means that results up to and including 8 processors run on a single node. With 64 processors we run 8 processors per node on 8 nodes. We use the term processor to refer to a processing element in a particular system. This could be either a thread or a standard process.

For parallelism within the nodes, we use OpenMP for Shray and Global Arrays. For Chapel we use the task parallelism it provides. UPC does not provide explicit multithreading, and recommends the user to start one task for each core, instead of one task for each node. Inter-node communication will then automatically use

shared memory for communication. For DGEMM we use one process per node anyway, and use the multithreaded version of our BLAS implementation. As the total communication volume scales with the number of processes used, this is the only way to do a fair comparison.

The large benchmarks have been run five times, the micro-benchmarks ten. The graphs show error bars of one standard deviation.

## Micro-Benchmarks

We benchmark two important parts of our DSM layer: the overhead of fetching a remote page through a SEGFAULT handler and the overhead in synchronisations.

**Page-Fetch Overhead** To benchmark how long it takes to fetch a remote page, we have one rank sum up the remote elements of a distributed array, as in Listing 5.8. This function is executed only by the first rank, so ShrayEnd(arr) marks the start of the remote data. We know that every $\frac{pagesz}{sizeof(double)}$ accesses, a SEGFAULT is going to be triggered, and Shray will load in that page. So for reference, we also implement this directly in GASNet, as in Listing 5.6. In that version we use a local buffer the size of a page to store the equivalent amount of data. The outer loop iterates over non-local ranks, the middle loop iterates over the pages, and finally the inner loop sums up the values on that page.

As the communication and local computation pattern is exactly the same, this should show the overhead we get from our SEGFAULT handler. To make sure we test the communication network and not the shared memory copy behaviour, we run this with one rank per node.

To put the results into perspective, we compute the achieved bandwidth. For reference: the OSU version 7.0 [179] point to point bandwidth test shows our system has a peak bandwidth of about 117 MB/s.

```
double reduce(double *arr, size_t n) {
  double sum = 0.0;
  for (size_t i = ShrayEnd(arr); i < n; i++) {
    sum += arr[i];
  }
  return sum;
}
```

Listing 5.8: Reducing the non-local part of an array in Shray

The physical page size on our system is 4096 bytes, but Shray allows us to use a virtual page size that is a multiple of this using the SHRAY_CACHELINE environment variable. We perform experiments for 1×, 10×, and 100× the system page size. From 10 runs on an 80 MB array we compute the sample variance $\sigma$. The results are presented in Table 5.2.

When comparing the GASNet and Shray figures, we see that the overhead of the SEGFAULT handling is between 1% and 7%, with the overhead getting smaller as we increase the virtual page size. We can also observe that the effectiveness of the transfers increases as the virtual page size increases. Using individual system

```
double reduce(double *A, size_t n) {
    double result = 0.0;
    int p = gasnet_nodes();
    size_t db_per_page = pagesz / sizeof(double);
    size_t blocksize = (n + p - 1) / p;
    double *buffer = malloc(pagesz);
    for (size_t j = blocksize; j < n;
            j += db_per_page) {
        int processor = j / blocksize;
        gasnet_get(buffer, processor,
                   A + j, pagesz);
        for (size_t i = 0; i < db_per_page; i++) {
            result += buffer[i];
        }
    }
    free(buffer);
    return result;
}
```

Figure 5.6: Reducing the non-local part of an array in GASNet, using Shray's communication pattern

Table 5.2: Achieved bandwidth in MB/s of reducing an array with page sizes 4 KiB, 40 KiB, 400 KiB, including the observed standard deviation

| Nodes | GASNet (4 KiB) | Shray (4 KiB) | GASNet (40 KiB) | Shray (40 KiB) | GASNet (400 KiB) | Shray (400 KiB) |
|---|---|---|---|---|---|---|
| 2 | 25±0.10 | 23±0.25 | 83±0.50 | 79±0.15 | 110±0.32 | 111±0.05 |
| 4 | 24±0.05 | 23±0.07 | 82±0.43 | 79±0.06 | 110±0.37 | 110±0.04 |
| 8 | 24±0.03 | 23±0.05 | 82±0.37 | 79±0.07 | 110±0.31 | 110±0.09 |

pages, we only achieve 20% of the machine's peak bandwidth; increasing it to 40 KiB already pushes this to 68% and with 400 KiB we get to 95%.

This shows clearly that larger virtual page sizes are preferable, implying that the overhead of Shray's
SEGFAULT-triggered page fetches is rendered negligible.

**Synchronisation Overhead** To benchmark our synchronisation system, we create an 8 GiB array, and then repeatedly synchronise. The average time per synchronisation in microseconds is shown in Table 5.3. We use one rank per node. For comparison the time of a GASNet barrier is given. The synchronisation cost is about 2×-3× the cost of a barrier. Note here that the owner-compute rule guarantees that no data exchange beyond two pages is needed during these synchronisations, making the cost constant in the size of the array. This is in stark

Table 5.3: Time to synchronise an array in microseconds for various number of nodes

| Nodes | GASNet barrier | $\sigma$ | ShraySync | $\sigma$ |
|---|---|---|---|---|
| 2 | 83.1 | 0.11 | 246 | 0.10 |
| 4 | 125 | 0.26 | 308 | 0.20 |
| 8 | 177 | 0.25 | 376 | 0.35 |

contrast to unrestricted DSM systems which inflict overhead in order to keep track of non-local changes, and which need to communicate non-local changes during such synchronisations.

## Dense Matrix Multiplication (DGEMM)

In this experiment we investigate weak scaling instead of strong scaling. The reason for this is that we evenly distribute the computation, but we need to access the entire matrix $B$, as defined in Subsection 5.4, regardless of the number of nodes. If we have $p$ nodes, that means each node has a computational load of $O(n^3/p)$ and a communication load of $O(n^2)$. So the algorithm is weakly scalable, but not strongly. For Shray we use the single BLAS call from Listing 5.5 with a page size such that each node has 100 pages of local data. UPC and Global Arrays use the blocked version given in Listing 5.6 while Chapel uses Listing 5.7. The initial size of the matrix is $2146^2$ elements. As we double the number of processors we also double the size of the matrix in bytes. For 64 processors we have value of 17280, which means we have $17280^2 \cdot 3 \cdot 8 \approx 6.67$ GiB of data. This value allows for reasonable benchmarking times with our setup. The results are shown in Figure 5.7. Both axis are logarithmic and the x axis displays the configuration it was run with.

Up to 8 processors all programs use the same optimised BLAS implementation, but except for Shray the matrices need to be copied out from distributed structures to non-distributed ones. This can lead to slightly worse performance. It is surprising that Chapel does not scale well going from 4 to 8 cores. We observed the same behaviour when stripping the program down to a single locale C = dot(A, B) operation, and verified that the correct BLAS library was indeed called. When going out of node, we are hampered by the slow 1 Gb/s interconnect as this a communication heavy benchmark. This explains the slowdown when going from 8 to 16 processors. Shray, UPC, and Chapel show the same scaling behaviour. Global Arrays scales more poorly. The remote fetch NGA_Get of Global Arrays shows similar performance to other remote fetches, so this is likely related to the access consistency. The results show that, for this limited setup, Shray is able to get equivalent performance characteristics compared to other frameworks.

## Conjugate Gradient

Conjugate gradient descent is an iterative method that approximates the solution of a linear system [106]. The computational heart of this algorithm is a matrix-

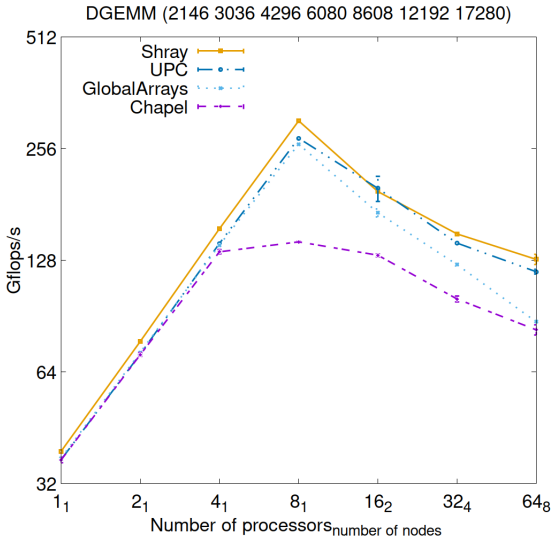



Figure 5.7: Dense matrix multiplication, weak scaling

vector multiplication. In practice, many matrices contain a lot of zeroes. Storing these increases the memory consumption, which may lead to scalability problems. It also introduces redundant computation in matrix-vector multiplication. A matrix containing a lot of zeroes ('a lot' is subjective) is called *sparse*. The Conjugate Gradient benchmark of [16] uses such a sparse matrix-vector multiplication. There are a variety of storage formats, this benchmark uses Compressed Row Storage (CRS) [41].

Using this storage format, the multiplication looks like Algorithm 11.

**Algorithm 11** Sequential sparse matrix-vector multiplication for the CRS data structure.

**Input:** $x$ : dense vector of length $n$.
**Input:** $A$ : sparse matrix of size $m \times n$.
**Output:** $y$ : dense vector of length $m$ such that $y = Ax$.

1: **for** $i := 0$ **to** $m - 1$ **do**
2: $\quad y_i := 0$
3: $\quad$ **for** $j :=$ row_ptr[i] **to** row_ptr[i + 1] - 1 **do**
4: $\quad\quad y_i = y_i +$ values$[j] \cdot x_{\texttt{col_ind}[j]}$

A challenging aspect of sparse matrix-vector multiplication is that we can not statically predict the access pattern. Indexing into the vector depends on the column index in the compressed matrix. As such, the PGAS implementations have to perform explicit checks on every access. The NAS benchmark of [16] provides multiple problem sizes. Class S is the smallest while class F is the largest.

Chapel's distribution contains a sequential implementation for this benchmark,

which we adapt to use multiple cores within a node with forall constructs, and on multiple nodes by using a block distribution on the dense space containing the sparse matrix. For the other versions we adapt a C version [198] that is an adaptation of the reference implementation of [16]. We keep their OpenMP parallelisation for intra-node communication for Shray and Global Arrays.

Figure 5.8 shows a comparison between implementations for problem size S. We have no figures for higher classes as they timed out for the alternatives to Shray. The axis are logarithmic for this example as well, meaning Shray is around 140 times faster than the second best at 64 processors. We see that Shray has the highest performance for all processor numbers. As was mentioned, the PGAS alternatives have to perform explicit checks on each access while Shray is free of this burden. Furthermore, as Shray fetches a page instead of a single element we paradoxically communicate less. This is because we get multiple cache-hits on one page. Fetching one element does not actually lead to 8 bytes of communication as Ethernet frames are between 64 and 1536 bytes. Profiling measurements show that for class S, Shray communicates 13 MB from node 2 to node 1 for 2 nodes / 16 processors total. Chapel in contrast performs 3810008 unblocking fetches, and 3043670 blocking ones. This adds up to a communication load between 438 MB and 10.5 GB, depending on the size of the ethernet frames. Global Arrays shows roughly similar characteristics to UPC and Chapel when going out of node and does 8010912 remote gets which is 0.512 - 12 GB, and we suspect that something similar happens with UPC.

Figure 5.9 shows the performance of Shray for all problem sizes up to class D. Larger sizes were not feasible on our cluster as each of the 8 nodes only comes with 16 GB of memory. Class C is around 700 MB, $D$ around 11 GB and $E$ around 113 GB.

We see that smaller problem sizes do better within a node, and larger ones do better when using multiple nodes. The irregular access pattern hurts performance for our cache-like Shray system, but also for the actual hardware caches. Smaller problem sizes need smaller caches, which is why they do better within a node. The number of synchronisations per iteration are fixed, though the number of flops increase with larger problem sizes. Therefore the performance hit when using more than one node is smaller for the larger problems. For class S, we communicate only 26 MB in total, but have 1270 barriers.

## 5.6 Related Work

A large body of related work exists. In the context of MPI, quite some work has been dedicated towards creating support for expressing re-occurring communication pattern on a higher-level of abstraction. Most notably, the work on collective operations in MPI [77, 112] tries to help programmers of distributed applications to more easily write more efficient code. While such approaches substantially ease MPI program development, they still require programmers to carefully decide about data distributions and to make communication explicit, albeit on a higher level of abstraction.

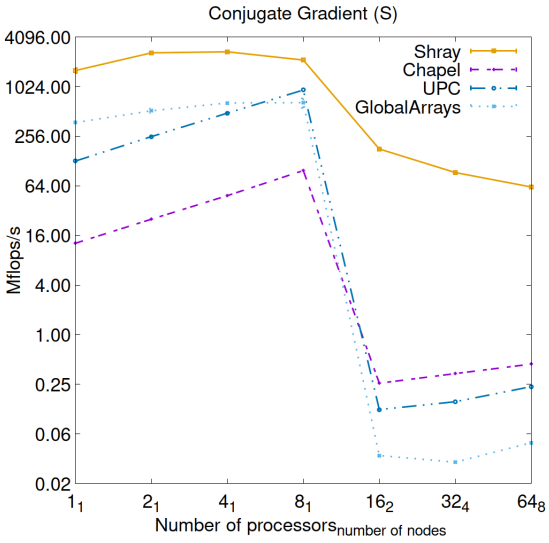



Figure 5.8: Conjugate Gradient class S

More closely to the work presented here is the classical work on DSM [75]. Several highly sophisticated DSM systems exist ranging from hardware systems (e.g. [166, 64, 154, 150]), over hybrid systems such as [148, 34] to pure software solutions [156, 133, 69, 163]. In contrast to oc-DSM, all these systems aim at non-restricted write operations. While this makes programming potentially easier than with the owner-compute restriction of Shray, it comes with the burden to statically or dynamically identify where write-operations have to be sent to, leading to various different memory consistency models. The most convenient consistency model from a programmer perspective is strict sequential consistency [151], where "the result of any execution is the same as if the operations of all the ranks were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its programmers". This is used by systems such as IVY [156]. The strict requirements of sequential consistency, however, render performance optimisations very challenging. Consequently, many ways to relax this consistency have been proposed, see [218] for a survey and comparison of some. All relaxed consistency models indirectly bring back some user attention towards the fact that data is physically distributed, and, more importantly, they pose a non-trivial overhead on the existing implementations.

More recently, novel interest in DSM has emerged, motivated by the observation that the gap between remote processor accesses and main RAM accesses has closed significantly [202]. While this promises better performance with as little as possible programmer restrictions, the need to establish memory consistency in a generic way still requires non-trivial machinery, increasing the complexity of the implementation alongside the need to dedicate a non-trivial amount of compute resources to this task.

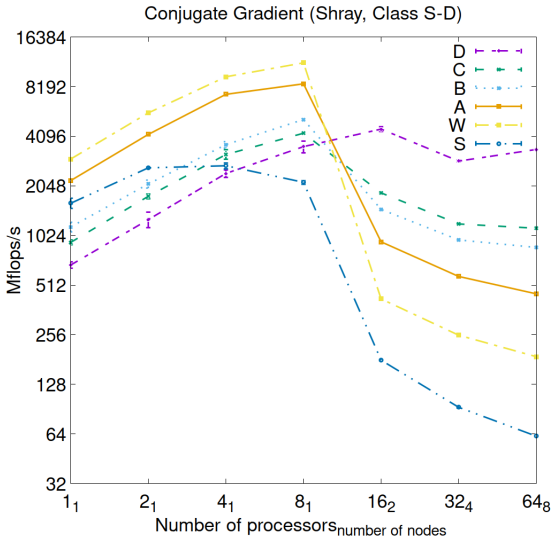



Figure 5.9: Conjugate Gradient up to class C for Shray

The performance challenges involved in establishing memory consistency have driven a lot of related work in the context of languages and libraries based on the idea of PGAS.

Dart [249], software for data assimilation research, and UPC [86], use global pointers for distributed structures. On read and write operations, the run-time calculates where to get the data from, and uses MPI's or GASNet's RMA functions for the actual communication. The UPC compiler contains several optimisations to resolve multiple references with a single communication operation [51]. As shown in Section 5.4, the choice in UPC to distinguish the distributed structures from normal ones can inhibit the use of library functions that are not PGAS-aware. Memory consistency in UPC is established through synchronisation barriers and through explicit locks. While Shray also requires synchronisation barriers, the distributions are completely orthogonal to external libraries as long as these are used in a way that ensures that all write operations are local. Here, Shray's ability to enforce a granularity for partitioning comes in handy as demonstrated at the example of matrix multiply in Section 5.4.

Similarly, Global Arrays [185] provides global synchronisations for ensuring consistency.

Futhark [104] is a functional array programming language that compiles to parallel architectures. Recently, a backend for distributed memory with MPI has been proposed [72]. This implementation stores all data on all nodes, does computation on a chunk of the data, and then gathers all the data again. This approach is not memory-scalable and incurs significant communication overhead. Shray may be a viable alternative as a compilation target.

## 5.7 Conclusion

This chapter presents a new point in the design spectrum between sequentially consistent DSMs and explicit PGAS systems with a strict separation of local and remote reads and writes. The new design, named oc-DSM, treats reads and writes differently. While reads can be used as if all data is stored locally, writes are restricted to purely local writes, hence the notion of ownership. As it turns out, this design has several interesting properties: Firstly, the restriction to purely local writes poses surprisingly little programming burden for several frequently used algorithms. Transforming a sequential program into a parallel one using Shray often boils down to replacing the memory allocation and to restricting loop ranges. In the chapter we show this in detail for matrix multiply and conjugate gradient. Secondly, an MMU-based implementation allows for a very elegant and lean thread-safe implementation. Our open-source implementation [216] comprises about 800 lines of code. Thirdly, the oc-DSM design allows for synchronisations that impose minimal overhead. The potential data-exchange that can arise from shared pages involves communication between each node and a maximum of the two neighbouring nodes, irrespective of how many nodes participate in total. Fourthly, using the MMU enables "normal" read operations. In combination with being thread-safe, this opens the door for using many libraries that have been developed with a sequential system in mind. Examples are the use of highly optimised BLAS libraries or multi-threading libraries / tools such as openMP. Finally, our evaluations of matrix multiply and conjugate gradient show that, despite of the lean implementation, surprisingly competitive performance can be achieved. In particular in the case of unpredictable access pattern such as in conjugate gradient, we see up to two orders of magnitude better performance in Shray than the performance we measured when using Global Arrays, UPC, or Chapel.

Given these very encouraging observations, we see several avenues to further improve on the idea of oc-DSM. The original idea for oc-DSM was based on the observation that several high-level array languages such as Futhark [104] or SaC [207] have a tight control over write operations in order to guarantee a correct parallel semantics. Code generators for these languages such as [162] should be able to target a Shray-based runtime system guaranteeing the absence of writes to non-owned addresses while liberating the programmer entirely from the necessity to think about how memory is being handled.

Besides targeting Shray with code generators, there are many opportunities to improve Shray itself. GASNet requires RDMA regions (the "owned" part of the array) to be registered at the start of the program, unless dynamic pinning is used. This does not support the type of memory management we do. MPI has a similar issue. So in order to support RDMA hardware, we would need to implement Shray in a lower level library that allows for dynamic registering of RDMA regions, such as *ucx* or *libibverbs*. Dynamic load balancing could be supported by shifting ownership boundaries dynamically. This would benefit many applications, in particular in the context of sparse linear algebra with unpredictable sparsity patterns. Further improvements could be achieved by experimenting with more sophisticated cache replacement strategies for Shray. The conceptual view of

a 2d block distribution as a higher-dimensional array is very similar to the approach of implementing multi-threaded matrix multiplication in the array language SaC. It would be interesting to see if a Shray backend can extend this technique to distributed memory.

# Chapter 6

# Generating Distributed Code from Array Languages

## 6.1 Introduction

Many kernels for solving numerical problems, such as the FFT [56], MultiGrid Methods [39], (preconditioned) BiCGSTAB [239] have a practical time-complexity of $O(n)$ or $O(n \log n)$. This means that the bottleneck for getting sufficiently large or accurate solutions is not time, but memory. This necessitates the use of a collection of connected computers, to not only make more computational resources available, but more memory as well. Such a system is usually called a cluster, and one of these computers a node.

For an algorithm to take full advantage of such a system, the total memory requirements must not increase too much in the number of nodes. We say a parallelisation of a sequential algorithm is *memory scalable* if the memory usage on each node is $O\left(\frac{n}{p} + p\right)$, where $n$ is the memory usage of the sequential algorithm, and $p$ is the number of nodes.

Usually programs for clusters are implemented in MPI [174], which is a library that allows the user to send messages between nodes. There is a large semantic gap between an MPI program and the mathematical description of functions, as we need to explicitly break-up objects into pieces to distribute over the nodes, and explicitly communicate.

Even for simpler hardware such as shared memory machines, this gap exists. For example, multiplying matrix $A \in \mathbb{R}^{m \times k}$ by matrix $B \in \mathbb{R}^{k \times n}$ yields matrix $C \in \mathbb{R}^{m \times n}$ defined by $C_{ij} = \sum_{p=1}^{k} A_{ip} \cdot B_{pj}$. To implement this, we need to allocate memory, specify a traversal order, and specify how we would like to parallelise the computation.

This semantic gap makes it difficult for scientists to write and maintain their code: in addition to their domain, they need advanced knowledge of the hardware they write code for. In an effort to close this gap, functional array languages have been proposed. These abstract away low-level details such as memory, explicit

parallelism and explicit communication. For example matrix multiplication in Single Assignment C (SaC) becomes as follows.

```
C = {[i, j] -> sum({[p] -> A[i, p] * B[p, j]})};
```

Listing 6.1: Matrix Multiplication in SaC

This allows the compiler to generate code for a variety of hardware, from a single source. These languages have proven to be able to generate code that is as efficient as handwritten code in a number of benchmarks, for CPU and GPU architectures [18].

Generating code for clusters is currently underdeveloped. To the best of our knowledge, automatic parallelisation for clusters in a memory-scalable way is not yet possible. In this chapter we show how we redesign the distributed backend of SaC to target the external library Shray. This gives us memory-scalability, performance improvements, and simplifies the compiler. We believe most of these ideas are applicable to any data-parallel language.

This chapter makes the following contributions:

- We show how to generate memory-scalable Shray code from SaC, integrated with multithreaded code generation.
- We provide a tentative performance evaluation using up to 1024 cores.

## 6.2 Background

### Shray

Shray [212] is a C-library that provides an easy way to rewrite a shared memory program to a distributed program. This is perhaps best explained by an example. Suppose we have a function that computes an array X and that we want to distribute this because X does not fit into a single machine.

```
double *init(size_t n, size_t m) {
    double *X = malloc(m * n * sizeof(double));
    #pragma omp parallel for
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n; j++) {
            X[i * n + j] = f(i, j);
        }
    }
    return X;
}
```

Listing 6.2: Single-node array computation

We would change the allocation function to ShrayMalloc, which takes the extent of the first dimension as an argument. After this function returns, we can use X as if it were allocated in shared memory. That means that the program

outside of initP does not need to be modified, except for changing free(X) to ShrayFree(X).

```
double *initP(size_t n, size_t m) {
  double *X = ShrayMalloc(m, m * n * sizeof(double));
  ...
}
```

Listing 6.3: Distributed allocation with Shray

Shray uses the SPMD paradigm, meaning that all $p$ processors will execute the same program, but on different data. The parallelism comes from each processor computing a part of X. Shray gives access to functions ShrayStart and ShrayEnd. These will return different size_t values between 0 and n, depending on the processor. Denoting $start_{id}$ and $end_{id}$ for the returned values at processor id, these form a partition

$$\bigsqcup_{id=0}^{p-1} [start_{id}, end_{id}) = [0, m).$$

That means that a parallelisation only has to restrict the first loop to the part of this partition that is associated with the processor.

```
double *initP(size_t n, size_t m) {
  ...
  #pragma omp parallel for
  for (size_t i = ShrayStart(X); i < ShrayEnd(X); i++) {
    for (size_t j = 0; j < n; j++) {
        X[i * n + j] = f(i, j);
    }
  }
  ...
}
```

Listing 6.4: Restricting a loop-nest with Shray

In fact, this is the only way of parallelising that is allowed: we say that an index $(i_1, \cdots, i_d)$ of some array $X$ is owned by a processor, if ShrayStart(X) $\leq i_1 <$ ShrayEnd(X). We require the owner of an index to compute the corresponding value. Non-owned values are only visible after ShraySync has been called, which cumulates in the final parallelisation of Listing 6.5.

For this chapter, we have reimplemented Shray on the low-level library UCX [214] to address the performance concerns outlined in Chapter 5. We will refer to this implementation as SHRAYonUCX [217]. As UCX is also used in common MPI implementations such as OpenMPI and MPICH, the new Shray version is MPI-compatible, and the generated code can be run as normal MPI program.

```
double *initP(size_t n, size_t m) {
  double *X = ShrayMalloc(m, m * n * sizeof(double));
  #pragma omp parallel for
  for (size_t i = ShrayStart(X);
            i < ShrayEnd(X); i++) {
    for (size_t j = 0; j < n; j++) {
      X[i * n + j] = f(i, j);
    }
  }
  ShraySync(X);
  return X;
}
```

Listing 6.5: Full parallelisation with Shray

## SaC

Single Assignment C (SaC) is a functional array language suitable for numeric computations. It allows for high-level specifications close to the mathematics, and the compiler can automatically parallelise code that has been expressed using concurrent constructs.

### Arrays and Reductions

SaC has one constructs for expressing concurrency. For example, in matrix multiplication, C = {[i, j] -> sum({[p] -> A[i, p] * B[p, j]})}, there are two computations that could be done in parallel: each value of C can be computed independently from each other, and the sum can be computed in parallel by computing subsums and then combining them.

Both of these constructions are cases of a *with-loop* hidden under a thin layer of syntactic sugar [210]. SaC only allows rectangular arrays without gaps, so we need a rectangular index set containing these generators, and a way to fill up the gaps. A *genarray* takes a *d*-dimensional shape *shp* which determines the index set as $[0, shp[0]) \times \cdots \times [0, shp[d - 1])$, and a default value to put at the holes. A *modarray* takes a different array, which determines the index set and the values at the gaps. The sum is an instance of a *fold*-with-loop: the expression is combined over the index set using some binary function, addition in the case of sum.

### Multithreaded Code Generation

SaC never generates nested parallelism. This requires us to know at any with-loop whether we are already executing in parallel. Rather than keeping a global state, the multithreaded backend generates two versions of each function: a sequential (ST) version, and a multithreaded (MT) version. Any with-loop in a MT-function is never parallelised. Parallel with-loops are lifted in their own functions (SPMD), and function calls within such an SPMD-function are done to the MT-variant.

```
x = with {
    (lb1 <= iv < ub1 step s1 width w1): expr_1(iv);
    ...
    (lbn <= iv < ubn step sn width wn): expr_d(iv);
}: genarray(shp, default);
y = with {
    (lb1 <= iv < ub1 step s1 width w1): expr_1(iv);
    ...
    (lbn <= iv < ubn step sn width wn): expr_d(iv);
}: modarray(x);
```

Listing 6.6: The two ways of creating an array in SaC: a genarray with-loop and a modarray with-loop.

Listing 6.7 shows how Listing 6.2 is expressed in SaC, and Listing 6.8 illustrates the generated code.

```
init(int m, int n) {
    X = {[i, j] -> f(i, j) | [i, j] < [m, n]};
    return X;
}
```

Listing 6.7: Array Computation in SaC

The parallel code generation partitions the index space over the available threads. In the case of fold with-loops, the thread-local results are accumulated before returning from the SPMD function. For this chapter, we have extended the multithreaded backend to partition over the first two dimensions.

#### State and Foreign Function Interface

The majority of a SaC program consists of the constructs described so far, however, these constructs do not suffice for tasks such as I/O.

Purely functional languages like SaC do not allow functions to have side-effects. Yet they still need to interact with the outside world. SaC uses uniqueness typing [3]. A unique object is an object that can only be used once. In the case of printing this may look like

```
new_stdout = print(stdout, "hello world!");
```

Here stdout, new_stdout are unique objects, and new_stdout represents the standard output after "hello world!" has been printed to it.

Furthermore, if an optimised library is available, we would like to use it. SaC allows this through a Foreign Function Interface (FFI). The data structures such a function uses may not be (straightforwardly) representable as arrays, such as a linked list. For this reason the FFI requires the user to provide a function to free and copy the data. The type of such an object is then denoted as HIDDEN.

```
double lift::SPMD(int i, int j) {
    X = { /* parallel */
          [i, j] -> f::MT(i, j) | [i, j] < [m, n]};
}

init::ST(int m, int n) {
    if (m * n > threshold) {
        X = lift::SPMD();
    } else {
        X = { /* sequential */
              [i, j] -> f::ST(i, j) | [i, j] < [m, n]};
    }
    return X;
}

init::MT(int m, int n) {
    X = { /* sequential */
          [i, j] -> f::MT(i, j) | [i, j] < [m, n]};
    return X;
}
```

Listing 6.8: Multithreaded Code Generation

## 6.3 Design and Implementation

The SaC compiler generates a C program using Shray and MPI in SPMD style. That means each node executes the same program, and in principle replicates the computations. The only exceptions to these are with-loops and functions dealing with state. Here each node still executes the same program, but they execute it on different parts of the index sets.

### With-Loops

The basic idea is simple: we extend the multithreaded backend, and reuse its machinery to decide on what to parallelize. We restrict the range of a with-loop using ShrayStart, ShrayEnd before passing it into the multithreaded scheduler. We know that parallel with-loops happen in ST functions for with-loops of a certain size.

Data-parallel with-loops must operate on memory allocated by ShrayMalloc. To this end, we replace the allocations in ST-functions by a different allocator that decides whether to use ShrayMalloc or malloc depending on the size of the array. We store this decision in the descriptor of the array. In the majority of cases, this gives the correct memory to SPMD functions. However, there are some edge-cases. For example, some builtin function may not support a distributed memory implementation, and the resulting non-distributed memory may be reused for a

with-loop that should be parallelised. For that reason we insert a builtin function distmemify that reallocates using ShrayMalloc if the allocation was not already distributed before SPMD-function calls. We do the opposite before sequential with-loops. This results in Listing 6.9.

```
double lift::SPMD(X_mem) {
    X = { /* parallel */
          [i, j] -> f::MT(i, j) | [i, j] < [m, n]};
}

init::ST(int m, int n) {
    X_mem = allocd(m * n);

    if (m * n > threshold) {
        X_mem = distmemify(X_mem);
        X = lift::SPMD(X_mem);
        ShraySync(X_mem);
    } else {
        X_mem = undistmemify(X_mem);
        X = { /* sequential */
            [i, j] -> f::ST(i, j) | [i, j] < [m, n]};
    }
    return X;
}

init::MT(int m, int n) {
    X = { /* sequential */
        [i, j] -> f::MT(i, j) | [i, j] < [m, n]};
    return X;
}
```

Listing 6.9: Distributed Code Generation

In case of fold-loops, the result is replicated on all nodes, so we do not insert (un)distmemify functions. Instead, we generate a fold_nodes built-in function after SPMD function calls, that does the reduction across nodes, similar to an MPI_Allreduce.

## Dealing with the External World

An external function can only produce observable effects if it takes a unique object. For this reason, we only execute those calls on node 0. The return values of this function are then broadcast to all other nodes with one exception: an external function may return a hidden type, which we cannot straightforwardly broadcast as we do not know the size, and because it may not be contiguous in memory. For that reason, hidden objects are also only valid on node 0, and we treat functions that take a hidden object the same way we treat functions that take a unique object.

## 6.4 Experimental Evaluation

To evaluate the effectiveness of our approach, we have benchmarked three applications. The first, Full Multigrid (FMG), can be considered a typical application. The runtime complexity is linear, and the application is bottlenecked by memory usage. FMG exposes several interesting features: grids varying in size, recursion, data-parallel operations, and reductions. The second, an N-Body simulation, has quadratic runtime complexity and is therefore not bottlenecked by memory usage. We use this to measure how well we use the computational power of multiple nodes. Finally, we look at matrix multiplication, a benchmark with significant data movement.

**Full Multigrid** The purpose of FMG is to solve the differential equation

$$\nabla u = f$$

on $[0, 2\pi]^2$ for $f : \mathbb{R}^2 \to \mathbb{R}$ a periodic function, i.e. $f(0, y) = f(2\pi, y), \quad f(x, 0) = f(x, 2\pi)$. We approximate $f$ with discretisations $F_n = \{f(i \cdot h, j \cdot h) \mid 0 \leq i, j < n\}$, where $h = \frac{1}{n}$. We implement an algorithm very close to the textbook [155] version, with as only modification that we use red-black ordering.

**N-Body Simulation** An N-Body simulation models the gravitational forces between $n$ bodies in $\mathbb{R}^3$. Such a body can be described by its mass $m$, position $p$, velocity $v$, and acceleration $a$. The update after a time $dt$ is described by

$$\begin{aligned} a_i &= \sum_{j=0}^{n-1} \frac{m_j(p_j - p_i)}{(\|p_j - p_i\| + \epsilon^2)^3}, \\ v_i &= v_i + a_i \cdot dt, \\ p_i &= p_i + v_i \cdot dt, \end{aligned}$$

where $\epsilon$ is a small softening value.

**Matrix Multiplication** We use the implementation of [219] with a minor modification: we do not (un)block as SaC does not support a distributed reshape yet. We set SHRAY_CACHELINE to 10000, such that every segfault requests 40 MB of data.

### Methodology

We do our experiments on the thin Rome partition of supercomputer Snellius. Each node is equipped with two 64-core AMD Rome 7H12 CPUs and 224 GiB of DRAM. Each node has one HDR100 ConnectX-6 interconnect, providing 12.5 GB/s of bandwidth between nodes. We use SHRAYonUCX at commit 27bfb4a861, GCC version 13.3.0, OpenMPI version 5.0.3, UCX version 1.16.0, and SaC at commit 1cd7110c6b. A copy of the benchmark and all used software is permanently available at [139].

The speed of FMG is limited by memory bandwidth, so we report performance in effective GB/s, e.g. total memory requirement in GB divided by time in seconds. The maximum bandwidth of DRAM is 215 GB/s per node. As the N-body benchmark is limited by compute, we report this performance in Gflops. The maximum for a sequential implementation is 19 Gflops. We also use Gflops for Matrix Multiplication.

We repeat each experiment 10 times, and we report both mean and standard deviation through error bars. We measure strong scaling for both applications, or in other words keep the work constant while increasing the number of compute nodes.

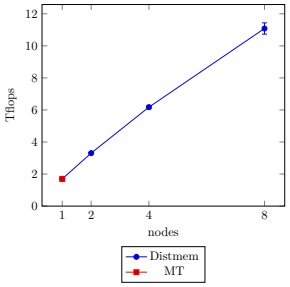
## Results

We see in Figure 6.1a that our implementation does very well on problems that are limited by computation power, obtaining 82 % efficiency compared to a multithreaded baseline. There is no overhead on local computations because Shray exploits the MMU for checking locality, and the cost of communication and Shray function calls is insignificant compared to the total computation required.

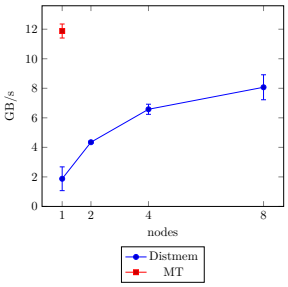
In Figure 6.1b, we report the effective bandwidth: required memory divided by time. The effective bandwidth on 8 nodes is 7.7 GB/s, so solving the largest Poisson problem that fits in the 2 TB of available memory takes $2048/7.7/60 \approx 4.4$ minutes. This is a reasonable runtime, so we can consider our goal of handling large problem sizes achieved.

Unfortunately, Figure 6.1b also shows a disappointing slowdown of $6.4\times$ when using one node. There are no segfaults or communication in this case, so the slowdown must be caused by ShraySync, ShrayMalloc, and ShrayFree. Profiling shows ShraySync and ShrayFree are responsible for less than 0.1 % of the total runtime. The calls to ShrayMalloc are significant, taking up a third of the total runtime. Over 99.9 % of this runtime is in registering memory with the coprocessor responsible for RDMA. The remaining slowdown must be caused by ShrayMalloc'd memory being slower to access. The mlx5 driver for the coprocessor uses Linux's Contiguous Memory Allocator through the DMA subsystem, so a possible explanation is that this memory is slower to use. Though we observe speedups when using more resources, the efficiencies are limited: 40 %–83 %.

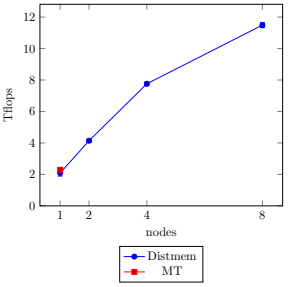
We have excellent scaling for the communication-heavy benchmark of Matrix Multiplication up to four nodes, with an efficiency of 93 %. At 8 nodes the efficiency drops to 69 %. This makes good on the promise of Chapter 5, where we blamed the poor performance of matrix multiplication on the limited bandwidth of the system, and the implementations' inability to make use of RDMA hardware. This has been addressed by using Snellius and the new SHRAYonUCX implementation. The limited scaling for 8 nodes is caused by 2.2 s of Shray overhead out of 9.6 s total runtime. Subtracting this gives a similar figure of 90 % efficiency. The 1D block distribution we use leads to a communication volume that is not asymptotically scalable [84], but it is not a practical problem at the scale of our experiments.




(a) Compute rate for N-Body simulation, $N = 2^{20}$, $iter = 10$.




(b) Effective bandwidth for FMG on a 32768 × 32768 grid.




(c) Compute rate for multiplying a 34560 × 40000 matrix with a 40000 × 40000 matrix.

Figure 6.1: Strong scaling experiments for three benchmarks on 1–8 nodes, with each node consisting of 128 CPU cores.

## 6.5 Related Work

The original distributed memory backend [162] uses a special memory allocator for distributed arrays, and translates indices to the corresponding distributed memory. This approach is not memory scalable, and the index translation comes with performance overhead and complexity in the compiler. We have also integrated the distributed memory backend with the multithreaded backend, reusing much of its infrastructure. As such, we need no global state to keep track of the execution mode.

Futhark makes the compiler instead of the runtime responsible for moving data [72]. In the fully general case it is unfeasible to statically decide what data is needed, so they duplicate the data structures across nodes, and synchronize by gathering the data. This is not memory scalable and incurs data movement linear in the size of the array for maps, Futhark's version of a genarray.

## 6.6 Conclusion

We have implemented a Shray-based backend for SaC. The basic idea behind the implementation is to restrict the computation of large enough data-parallel arrays. This achieves our goal of creating a memory scalable implementation.

In the hardware and software configuration we tested, there is a constant factor slowdown to accessing random access memory that is remotely accessible. This can be amortized in applications with a significant number of operations per memory access. The overhead of Shray-primitives we generate is significant, but constant in the problem size. These two factors explain why we obtained limited strong scaling in some benchmarks, but they should not affect weak scaling. In this chapter, we were unable to perform weak scaling experiments because SaC does not yet support 8 B indices. Nonetheless, SaC can now compute tasks that are too

large to fit in the memory of a single computer, and the design is scalable.

## Future Work

Most importantly, we need to increase the size of array shapes and internal indexing from the default 4 B int to an 8 B so that we can index large arrays.

The slowdown on ShrayMalloc'd memory is disappointing, but outside of a Shray's scope to fix. Improvements on the side of UCX and RDMA hardware will carry over to better results for Shray.

To use SaC in larger applications, we should make the FFI aware of distributed memory. This is feasible as SHRAYonUCX is compatible with MPI.

# Chapter 7

# Multi-GPU Code Generation for Out-Of-Core Problems

## 7.1 Introduction

Graphics processing units (GPUs) are processors that have higher peak compute rate and bandwidth compared to CPUs of comparable price and power consumption. This makes them attractive for a range of domains requiring extensive computation, such as physics simulations, financial applications and deep learning [88, 195, 126].

Obtaining the theoretic peak performance is challenging. The reason for this is that GPUs are not stand-alone systems but accelerators that are specialized to specific workloads. They need to be programmed differently, and have their own memory which needs to be coordinated with the CPU and RAM (also called the *host*). A straightforward way to increase the theoretic computational power of a system is to add multiple GPUs. However, realising this performance becomes difficult as the different GPUs need to be coordinated. An especially complicated class of problems are those that do not fit in the memory of a single GPU. We refer to these problems as being *out-of-core*. For an out-of-core problem we need to coordinate memories of different GPUs as well, making this a form of distributed computing, with all its associated challenges.

Making effective use of multiple GPUs requires the programmer to have extensive knowledge of the hardware in addition to the problem domain. A large body of research tries to shift this burden from the application programmer to compiler writers by generating low-level GPU code from a high-level language. In particular, functional approaches through languages such as Accelerate [45], Futhark [104] or SaC [207] or other array languages such as APL [122] show GPU performance that is competitive with hand-written counterparts [169, 9, 99, 115]. The related work uses extensive static analysis, sometimes in combination with

language extensions, to coordinate the GPUs.

The main idea behind this chapter is that we can shift the burden even further, by making the runtime instead of the compiler responsible for data-movement. This way only minimal changes to the compiler are necessary to support using multiple GPUs on out-of-core problems.

The mechanism that makes this possible is *unified memory*: one virtual address space that can be accessed by all GPUs and the CPU. The GPU driver and hardware are then responsible for maintaining a coherent view of memory between GPUs and the host. This approach is typically associated with bad performance. A reason for this is that the runtime needs to be advised on access patterns to work efficiently. In many languages this is hard to do, but by working with a functional language we have the strong semantic guarantees necessary to generate this advice.

This chapter makes the following contributions.

- We propose a code generation scheme that can use multiple GPUs and handle out-of-core problems. It constitutes a light-weight extension of a pre-existing code generation scheme for unified memory [238];
- We show the effectiveness of our approach by implementing this scheme in the language Single-Assignment C;
- We show how memory advice can overcome performance challenges typically associated with unified memory.

## 7.2 Background

Single-Assignment C (SaC) [208] generates CUDA code, a language for programming GPUs. We use this language to explain some GPU programming concepts, and briefly explain how SaC generates code.

### GPU Programming

Computations are done through special functions called *kernel*s. These contain a small computation such as "compute $t + 1$ and store it in index $t$", similar to a loop-body in C (Listing 7.1).

```
void f(int *x, int n) {
    for (int t = 0; t < n; t++) {
        x[t] = t + 1;
    }
}
```

Listing 7.1: Array initalisation in C

The programmer specifies for which $t$ this is computed at the call site. For example, to compute the set $X = \{t + 1 \mid 0 \leq t < n\}$, the kernel could be as follows.

```
__global__ void f(int *x) {
    int t = blockIdx.x * blockDim.x +
            threadIdx.x;
    x[t] = t + 1;
}
```

The initalisation of t (the *thread*) is complicated because there is not one canonical way to map computations to the hardware. Instructions are always executed in groups of threads, typically 32, similar to SIMD instructions on a CPU. Execution units on a GPU are grouped into *streaming multiprocessors* (SMs), which also includes cache and control logic. The threads are divided into groups called *blocks* of blockDim.x threads, and then divided over the SMs. The index local to one such block is stored in threadIdx.x, and the block index by blockIdx.x. Increasing the number of threads per block may help to hide latency, similar to hyperthreading in CPUs. However, it also decreases the number of blocks which can have a negative effect on scaling and load balancing. There is no choice that is optimal for all problems. The number of threads per block is limited by the hardware, typically 1024.

We can specify the mapping to hardware—so the number of threads per block blockDim.x and the total number of threads—between triple angular brackets at the call site. As a concrete example, suppose we want to use 1024 threads per block. Then we need $n/1024$ blocks, and call f as follows

```
f<<<n / 1024, 1024>>>(x);
```

We use *thread space* for referring to the shape $[n/1024, 1024]$.

The variables end with .x because they are actually three-dimensional structs with also .y and .z members. In this case all .y and .z members are 1, but for indexing higher-dimensional arrays it may be convenient to arrange the threads in a higher-dimensional grid. For example, 1024 threads may also be arranged in a $32 \times 32$ or $16 \times 16 \times 4$ grid.

## Unified Memory

Managing multiple address spaces requires extra work for the application programmer. Systems from personal computers to the world's largest supercomputer (Frontier, as of 2024) support abstractions that provide the programmer with a single virtual address space that can be accessed by the CPU and GPU. Depending on the vendor and exact implementation, these systems are called *unified memory*, *managed memory*, or *unified virtual addressing*. We will be using the term unified memory.

The implementation of unified memory in CUDA is not publicly accessible, but it seems that by default a page of memory can be accessed by only one processor — either a GPU or the CPU — at a time, and if a page is accessed that is not present in the respective processor's memory, a page fault occurs [52]. These page faults result in on-demand page migration, which can lead to mixed results [152, 137, 238]. This behaviour can be changed by giving the runtime *memory advice* on how

the memory is used. For example the CUDA option *cudaMemAdviseSetReadMostly* has the device create a local copy.

### Single-Assignment C

SaC is a functional array language for numeric programming. It tries to be as close to the mathematics as possible. For example, the array $X = \{t + 1 \mid 0 \leq t < n\}$ is specified as X = \string{[t] -> t + 1 [0] <= [t] < [n]}|. These array specifications, called *tensor comprehensions*, expose concurrency: each element can be computed independently from the others. The user can specify a compilation target, which the compiler sac2c will use to generate parallel code. For a GPU target, sac2c will generate a kernel from this specification. This kernel associates the *index set* [n] to the thread space [n / 1024, 1024], in some way we can view as a black box for the purpose of this chapter [123].

We can specify different expressions on different subsets of this index set, as illustrated in Fig. 7.1 and Listing 7.2. In the related work such a subset is also called a partition. In Section 7.3 we will talk about a different mathematical notion of a partition, so to avoid confusion we will use subset in this chapter.

The compiler has two GPU related targets. The first, cuda, explicitly defines arrays on the GPU and host. The necessary transfers between the two are then done explicitly. The second, cuda_man, uses unified memory instead (also called *man*aged memory). This makes memory transfers implicit: pages will be migrated over when they are touched. The cuda target has been observed to be more performant [238].

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $f(0)$ | $f(1)$ | $g(2)$ | $g(3)$ | $f(4)$ | $f(5)$ | $g(6)$ | $g(7)$ |

Figure 7.1: Array specified by Listing 7.2

```
{[i] -> f(i) | [0] <= [i] < [8]
                width [2] step [4];
 [i] -> g(i) | [2] <= [i] < [8]
                width [2] step [4]}
```

Listing 7.2: Array specified by two index sets

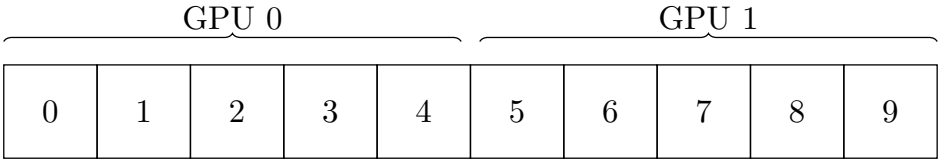
## 7.3 Design

To keep the implementation lean, we only make minor modifications to the existing target using unified memory and one GPU [238]. We need to implement a way to divide the work, and appropriately instruct the CUDA runtime on the memory access patterns.

## Work Distribution

In SaC, the index subsets may overlap, which requires us to process them sequentially. For this reason we do not distribute the subsets over the GPUs, but split up each subset individually.

We need to split up such an index set into non-overlapping pieces that together make up the original set (also called a *partition*). We do this by splitting along the first dimension, as illustrated in Fig. 7.2. To be more precise, for the simple case where the index set is of the form $[n]$, we give GPU $g$ rows $g \cdot b$ (inclusive) to $\max((g+1) \cdot b, n)$ (exclusive), where $b = \lceil \frac{n}{G} \rceil$, the smallest integer $b$ such that $n \leq Gb$. This choice minimizes the maximum number of rows per GPU, so for regular computations this gives optimal load balance.




(a) 1D distribution

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| GPU 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| GPU 1 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

(b) 2D distribution

Figure 7.2: Distribution of arrays over two GPUs.

### General Index Subsets

The most general way of specifying an index set in SaC is with a step and width. We need only minor modifications to the previous scheme. In set notation, an index set [l] <= [i] < [u] step [s] width [w] is equal to $[l, u) \cap S_{slw}$ where $S_{slw} = \{l + is + j \mid i \in \mathbb{Z},\ 0 \leq j < w\}$.

By distributing the intersection over the union, it becomes clear that a partition without strides over $G$ GPUs, say

$$[l, u) = \bigcup_{g=1}^{G} [l_g, u_g),$$

also gives a strided partition

$$[l, u) \cap S_{slw} = \bigcup_{g=1}^{G} ([l_g, u_g) \cap S_{slw}).$$

For $l_g \equiv l \bmod s$, we have $S_{sl_gw} = S_{slw}$. So under this restriction the strides and widths are the same.

In a similar vein to the simple case $[l, u) = [0, n)$, we set $l_g = l + gb$, $u_g = \min(l + (g + 1)b, u)$ for some $b$. To satisfy $l_g \equiv l \bmod s$, we must take $b$ of the form $sk$. The best load balance is now obtained by finding the smallest $b$ such that $bG \geq u - l$, which is $\lceil \frac{u-l}{sG} \rceil \cdot s$.

**Example 2.** *For index set [2] <= [i] < [1000] step [6] width [2] and two GPUs, we have $b = 504$. This results in index sets [2] <= [i] < [506] step [6] width [2] for the first GPU, and [506] <= [i] < [1000] step [6] width [2] for the second.*

### Implementation of Work Distribution

The partition of Fig. 7.2a corresponds to rewriting the SaC code

```
{[i] -> i | [0] <= [i] < [10]}
```

to

```
{[i] -> i | [0] <= [i] < [5];
 [i] -> i | [5] <= [i] < [10]}
```

It is therefore tempting to also do this in the implementation. However, this leads to unnecessary code duplication as one kernel per GPU would be generated. Furthermore, the number of GPUs would need to be known at compile-time.

To avoid this, we reuse the thread space mechanism. This mechanism must know what the index subset is, so parameters describing the subset must be passed into the kernel (at the very least for cases where these parameters are not known at compile-time). This means we can restrict the execution of the kernel to the subsets defined above by only manipulating the call site, which allows us to treat the kernel generation, thread space mechanism included, as a black box.

We manipulate the call site by inserting some code around the kernel call at the very end of the compilation process, when we generate the CUDA code, which we sketch in Fig. 7.3. The variables l0, u0 correspond to the lower and upper bound in the first dimension. In the loop we replace these by $l_g$ and $u_g$. To execute a kernel on a different GPU, we first call cudaSetDevice(g) where g is an index for that GPU. Then we call the kernel the same way as if we had one GPU. While the CPU will immediately continue after calling the kernel, the GPU has only finished computing after a call to cudaDeviceSynchronize().

## Memory Advice

An initial implementation of this code generation scheme gave poor results. For example, distributing the work of Listing 7.3 over two GPUs naively leads to a 6× slowdown compared to using a single GPU.

```
double[m, k] matmul(double[m, k] A,
                    double[k, n] B)
{
```

```
int old_l0 = l0;
int old_u0 = u0;

for (int g = 0; g < NUM_GPUS; g++) {
  /* Set l0, u0 to lg and ug */
  l0 = ...
  u0 = ...
  cudaSetDevice(g);
  /* The following three lines are
     unmodified from the
     cuda_man target. */
  grid = ...
  block = ...
  kernel<<<grid, block>>>(l0, u0, ...);
}

for (int g = 0; g < NUM_GPUS; g++) {
  cudaSetDevice(g);
  cudaDeviceSynchronize();
}

l0 = old_l0;
u0 = old_u0;
```

Figure 7.3: Kernel call distributed over NUM_GPUS GPUs

,

```
    C = {[i, j] -> sum({[p] ->
                A[i, p] * B[p, j]})};
    return C;
}
```

Listing 7.3: Matrix Multiplication in SaC

The entire jth column of B is needed to compute the jth element of a row of C. As discussed in Section 7.2, the memory of B can only be on one GPU at a time by default. Whatever GPU accesses the memory first will get the memory page, and the other GPU will have to wait until it can be migrated back. This serializes the computation between the two GPUs. Instead, it would be better if both GPUs created a copy of this page. This eliminates contention on reads, at the cost of making writes to the same page by multiple GPUs more expensive. This never happens in the code we generate because we only read from B. We can instruct the runtime to use this copy-on-read scheme by giving the cudaMemAdviseSetReadMostly memory advice.

This is not necessary for A. To compute row i of C, only row i of A is needed. As both C and A have m rows, the distribution over GPUs is the same. The data

that has to be accessed is already present on that GPU, so there is no contention between accesses.

Unfortunately, adding memory-advice has overhead, so we cannot insert it everywhere. In the example Nine-Point Stencil, whose most important computation is in Listing 7.4, adding memory-advice leads to a 2 – 5× slowdown compared to not inserting the memory advice.

```
{[i, j] -> w[0, 0] * x[i - 1, j - 1] +
           w[1, 0] * x[i    , j - 1] +
           w[2, 0] * x[i + 1, j - 1] +
           ...
           w[2, 2] * x[i + 1, j + 1];
  | [1, 1] <= [i, j] < [m - 1, n - 1];
...}
```

Listing 7.4: Sketch of the most computationally intensive part of a nine-point stencil.

This case is slightly more complicated than the access of A in Listing 7.3. The last row of GPU 0, which is the first row of GPU 1 is accessed by both GPUs. By GPU 0 as x[i + 1, ...] and by GPU 1 as x[i − 1, ...] . Except for the first and last row, all required data is already present on that GPU. We call this a *local* selection. The contention on these two rows is not enough to offset the cost of inserting the memory advice.

We can generalize this by looking at how the selection vector (for instance [p, j] in the case of B in Listing 7.3) relates to the index vector of the parallel tensor comprehension ([i, j] in both examples).

Let us write [iv1, ..., ivd] for the selection vector, and [jv1, ..., jvd] for the index vector. If jv1 is equal to iv1, the selections are all local, and we do not have to insert the advice. Suppose our GPU has rows $l$ to $u$. If iv1 is equal to jv1 + c for some constant c, then the accesses are local for rows $l+c$ to $u-c$. This is most accesses for small c. For this reason, we mark accesses for which iv1 is of the form jv1 + c. Given the functional nature of SaC we can then check for all relatively-free variables of tensor comprehensions whether all accesses are marked. If that is the case, we do not give memory advice. Otherwise, we mark them as read-mostly.

## 7.4 Performance Evaluation

We evaluate our approach in the style of distributed computing: we look at how much faster our code gets when adding more GPUs. The runtime of using one GPU divided by the runtime of using multiple GPUs is called the *speedup*. We expect at most a $G\times$ speedup when using $G$ GPUs. To measure how close we get to this, we also compute the *efficiency*: speedup divided by the number of GPUs. The closer this is to one, the better our system scales.

## Evaluation Platform

Scientific computations typically use double precision, but this is poorly supported in consumer-grade GPUs. To capture this, we use two different evaluation platforms. The first node (*icis*) has a similar double precision to bandwidth ratio as a consumer-grade GPU. The second node (*snellius*) has first-rate support for double precision. The hardware and software specifications are given in Table 7.1.

We use the SaC compiler at commit *f5890fa* with flags *-gpu_mapping_strategy jings_method_ext* for all benchmarks. Additional flags for a specific benchmarks are mentioned in their description. We used the standard library at commit *78c74b2*.

| | icis | snellius |
|---|---|---|
| GPU | RTX A6000 (2x) | A100 (4x) |
| VRAM (GB) | 48 | 40 |
| Peak Bandwidth (GB/s) | 768 | 1560 |
| Peak FP64 (GFLOP/s) | 604.8 | 9746 |
| C compiler | gcc 11.3.0 | gcc 12.3.0 |
| NVCC | 12.0.14 | 12.1.105 |
| NVLink bandwidth (GB/s) | 14 | 25 |
| Number of NVLinks | 4 | 4 |

Table 7.1: Overview of the test systems, hardware and software.

## Methodology

We consider three benchmarks, two with a high ratio of computations to array size (Matrix Multiplication and N-Body Simulation), and one with a low ratio (Nine-Point Stencil). We use floating point operations per second (*flops*) as a measure of performance for those with a high ratio, and bandwidth for the benchmark with a low ratio.

We evaluate the performance on a range of problem sizes, shown in Table 7.2. Class *D* of Matrix-Multiplication is out-of-core on both systems, the rest of the experiments are in-core. The reason for the small number of out-of-core examples is that SaC uses 32-bit integers for shapes, which limits the problem sizes.

We measure the costs inherent to the computation, that is the kernel-execution time, allocation of the result, and any inter-GPU communication. We do not measure any CPU-GPU transfer time of input and output because this may not be needed, for instance when the data is created and consumed on the GPU. The exact implementation and instructions for reproducibility are available [24]. We have used pragmas to manually reduce the number of threads per block for N-Body.

We compute and efficiency compared to the performance of our fastest, manually orchestrated target *cuda*. The efficiency for an out-of-core run is compared to the performance of the *cuda* target of the closest in-core class.

| Class | Matrix Multiplication | N-Body | Stencil |
|---|---|---|---|
| A | 10.1 | 0.010 | 4.29 |
| B | 19.7 | 0.021 | 9.66 |
| C | 38.3 | 0.042 | 17.2 |
| D | 51.0 | 0.084 | 34.0 |

Table 7.2: Overview of the memory footprints in GB.

## Matrix Multiplication

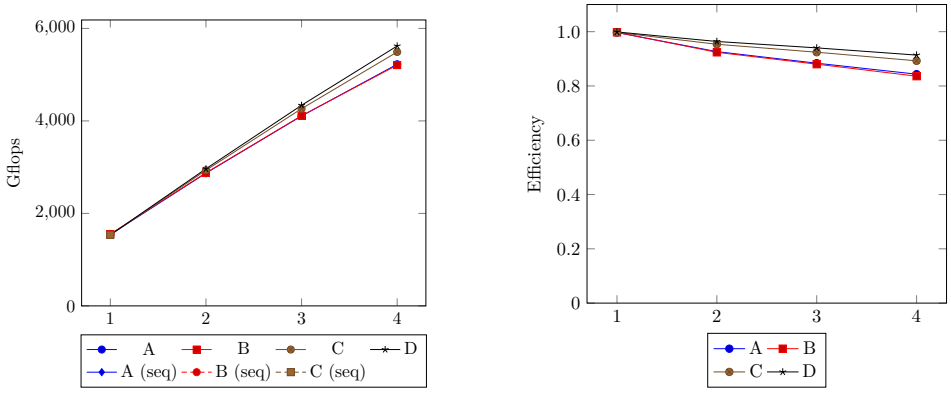
A matrix multiplication computes the composition of linear maps, making it a fundamental operation in linear algebra with applications in many areas. A few examples are linear programming, cryptography and deep learning [55, 50]. The implementation, Listing 7.3, is straightforwardly translated from the mathematical definition

$$C_{ij} = \sum_{p=1}^{k} A_{ip} \cdot B_{pj}$$

for $m \times k$ matrix $A$ and $k \times n$ matrix $B$.

Data from $A$ and $B$ are reused $k$ times. It is well-known that algorithmic changes that improve the temporal locality of accessing A and B can speed up the computation, but we refer this to future work.

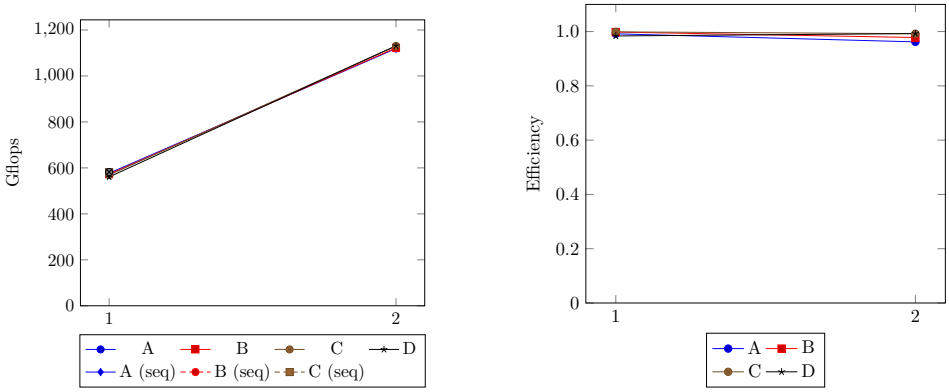
This computation takes $2mn(k-1)$ flops. The matrices $A$ and $C$ are perfectly distributed over the GPUs, but all GPUs need to access $B$. That means we need $O(n^2)$ data-movement and $O(n^3/G)$ computation for multiplying two $n \times n$ matrices on $G$ GPUs. The communication does not scale, but for large $n$ this term is dominated by the computation. So we expect better efficiencies for large $n$.




(a) Compute rate

(b) Efficiency

Figure 7.4: Matrix Multiplication on Snellius.




(a) Compute rate (b) Efficiency

Figure 7.5: Matrix Multiplication on Icis.

This is supported by the results of Fig. 7.4, where we see efficiencies ranging from 84% for classes A and B, to 91% for class D on four GPUs. The efficiencies are even higher in Fig. 7.5, where they range from 95% to 98%. The A6000 has a much higher bandwidth to FP64 ratio than the A100, so data-movement contributes even less to the compute rate.

It is interesting that class D does not seem to suffer from being out-of-core. This is likely caused by the unified memory overlapping this extra communication with computation [25].

## N-Body Simulation

The all-pairs N-body algorithm computes how the position and velocities of a system of bodies affect each other through gravity. In larger applications this exact algorithm is typically used to compute the forces between bodies that are near each other, and is combined with an approximate algorithm for bodies further away [20].

The main structure of this algorithm is described in Fig. 7.6. We have $[N, 3]$ arrays pos, accel, and velocity to represent vectors in three-dimensional space, and an array of shape $[N]$ for the masses.

```
accel = {[i] -> sum{[j] ->
          acc(pos[i], pos[j], mass[j]}};
velocity = velocity + accel * dt;
pos = pos + velocity;
```

Figure 7.6: Structure of nbody-simulation in SaC. The function acc computes the acceleration caused by the force of particle $j$ on particle $i$.

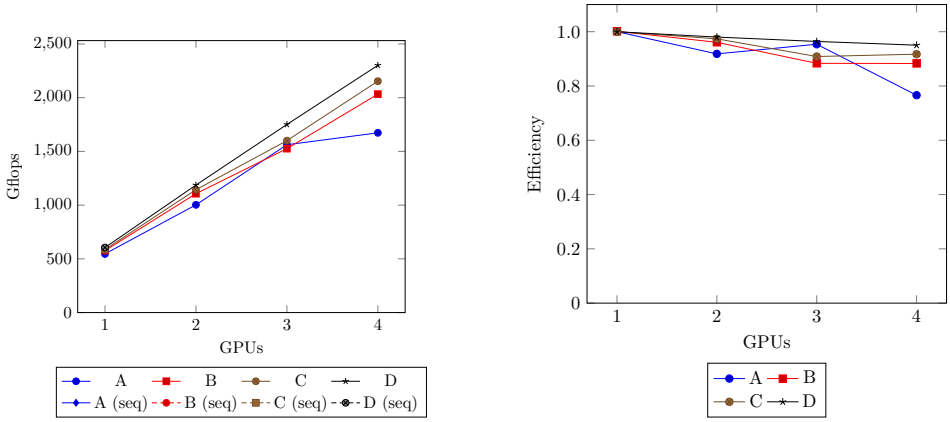
, ,

Most time is spent in the $O(N^2)$ computation of accel. The total cost is $20N^2 + 12N$ flops per iteration.

The performance characteristics are very similar to Matrix Multiplication, but even more pronounced. The computation is evenly distributed: for $G$ in total, each GPU does $O(N^2/G)$ computation per iteration. All bodies need to be replicated on each GPU, so that is $O(N)$ communication per iteration. In contrast to matrix-multiplication, this computation is typically in-core.

As expected, we see high efficiencies in Fig. 7.8, from 95% to 103%. This is the only benchmark where we see the efficiency exceeding 100%. This cannot be explained by variability in the measurements as the standard deviation is about 0.3%. The A6000 has 6 MB of L2 cache. Class $A$ does not fit in the L2 cache of one GPU, but does fit in the combined cache of both. We suspect this is the cause.

In Fig. 7.7 we have a high efficiency of 95% for the largest problem class $D$. For the smallest problem class $A$ this drops sharply for four GPUs, to 77%. The A100 has 6912 execution units per GPU, so 24768 in total. As problem class $A$ has 20480 bodies, we can keep only 83% of these units busy.




(a) Absolute performance

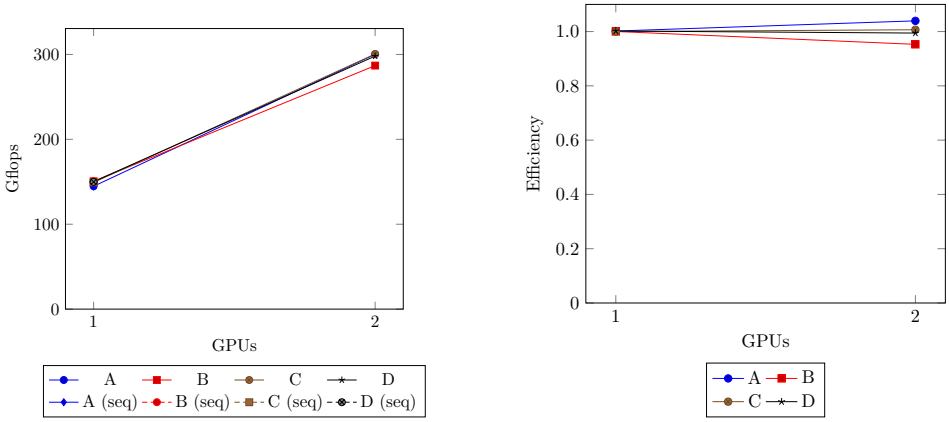
(b) Efficiency

Figure 7.7: N-Body simulation on Snellius.

## Nine-Point Stencil

The final algorithm we benchmark is a 2D iterative nine-point stencil computation. This updates each point in a 2D grid with a weighted averages of its nine neighbours, where we wrap around the edges. This is a common pattern in finite-difference methods for solving partial differential equations [19].

More recently, it has found application in convolutional neural networks [236, 223]. In contrast to the other two algorithms, each iteration does a small number of computations on each input element.




(a) Absolute performance

(b) Efficiency

Figure 7.8: N-Body simulation on Icis.

```
double[d:shp] relax(double[d:shp] x, double[d:wshp] w)
{
  return {iv -> sum({jv -> w[jv]*x[mod(iv+jv-wshp/2,shp)]})
                | iv < shp};
}
```

Listing 7.5: Stencil computation in SaC.

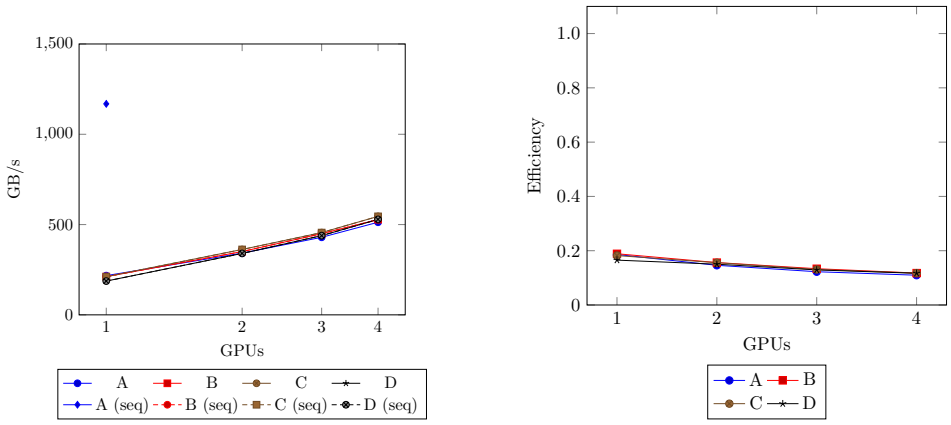
This tensor comprehension has nine index subsets because of the wrapping around the edges. In Listing 7.4 we have shown the generated code for the most computationally expensive one. Specifying these nine subsets is cumbersome, so we have used the more generic code of Listing 7.5. This can handle analogues in any dimension d, and also any shape wshp of weights. For example, if we want to take a weighted average of points at most two removed (Manhattan-distance) in three dimensions, we would have d = 3 and wshp = [5, 5, 5] [1]. The compiler is able to recognize the mod operator and generate the index subsets [237].

We use the flag −maxwlur 9 to ensure the weighted average of nine points in the inner computation is unrolled. This takes $17n^2$ flops per iteration for an $n \times n$ array, but it is more common to measure bandwidth as data-movement is the limiting factor. This is $16n^2$ bytes per iteration.

For $G$ GPUs, we need to do $O(n^2/G)$ data-movement within a GPU, and $O(n)$ data movement between GPUs. Therefore, we expect good results, similar to Matrix Multiplication and N-Body. However, Fig. 7.9 and Fig. 7.10 show no improvements over the cuda-baseline.

This is caused by a high cost for the first write to unified memory. When increasing the number of iterations, in Fig. 7.11 and Fig. 7.12, we do see speedups.

We can model the execution time for $T$ iterations as $Tx + c$, where $x$ is the cost per iteration, and $c$ the cost for allocation, deallocation and constant overheads.




(a) Bandwidth

(b) Efficiency

Figure 7.9: Stencil on Snellius, 10 iterations

From the measurements for $T = 100$ and $T = 10$, we can solve for $x$. The bandwidth based on this number is graphed in Fig. 7.13 and Fig. 7.14. Here we have efficiencies between 48% and 81%. The efficiencies are lower than the other two benchmarks, especially for the smaller problem classes. This is because the synchronisation cost is not negligible in this benchmark. Each iteration takes 0.009 seconds for problem class D on Snellius. For comparison, in Matrix Multiplication we have one synchronisation every 35 seconds, and in N-Body one synchronisation every 96 seconds.

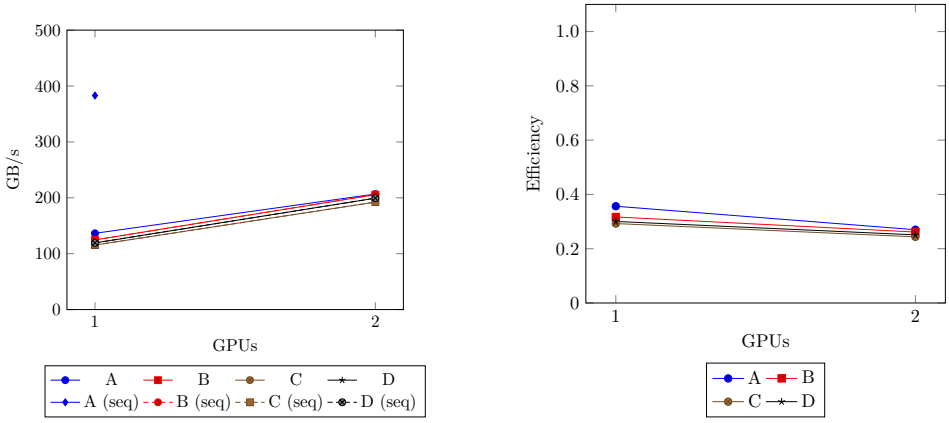
## Effect of Memory Advice

We can only investigate the result of memory advice on small problems and few GPUs because we quickly run into time limits. We have done measurements on Snellius for two GPUs. For Matrix Multiplication we take three $1024 \times 1024$ matrices, for N-Body we take 32768 bodies and 10 iterations, for Stencil we take a grid of size $1024 \times 1024$ and 100 iterations. We divide the performance of either always inserting memory advice or never inserting memory advice by the runtime of our optimisation in Table 7.3.

| | **Matrix Multiplication** | **N-Body** | **Stencil** |
|---|---|---|---|
| Always | 0.93 | 0.94 | 0.20 |
| Never | 0.09 | 0.72 | 1.00 |

Table 7.3: Speedup of always giving advice, never giving advice compared to our memory advice optimisation.

We see that always inserting advice gives good results for Matrix Multiplication and N-Body, but poor results for Stencil. If we never insert advice, the situation is reversed. Our optimisation makes the best choice in all three benchmarks. It




(a) Bandwidth

(b) Efficiency

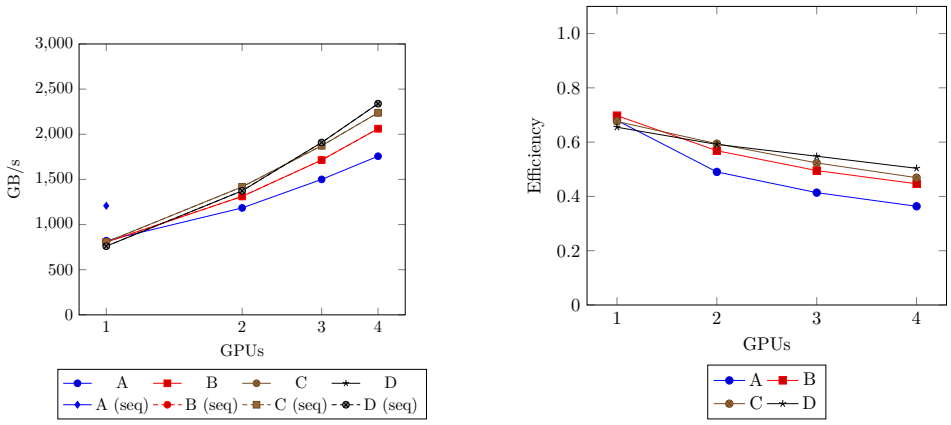
Figure 7.10: Stencil on Icis, 10 iterations

is even a little bit faster in Matrix Multiplication and N-Body because we do not insert the advice for arrays where we do not need to.

## 7.5 Related Work

Several studies have investigated compiling high-level specifications in functional array languages to multi-GPU code with manually-allocated data. One work presents envisioned compiler and runtime systems extensions for SaC for leveraging multiple multi-core CPUs and multiple GPUs without requiring additional programmer effort [67]. A new distributed-variable type is proposed to enable partial transfers of manually-allocated arrays between host and device. Furthermore, control structures are suggested to keep track of which parts of an array are present where, essentially implementing a cache coherency protocol. The authors evaluate a hand-coded prototype on a two-dimensional five-point stencil computations with two different combinations of parameters using GTX 580 GPUs. Their experimental results find that assigning almost all the workload to the GPU leads to the best results, and achieve a nearly two-fold speedup with two GPUs, but run into significantly less improvement with three or more GPUs.

The next study extends the Futhark compiler with a new internal streaming operator called the *husk* operator [113]. The author introduces a new type and semantics to further present transformations for the map and reduce second order array combinators of Futhark, essentially allowing an intermediate Futhark program to expose opportunities for distributable data and operations. The study then also extends Futhark's CUDA C backend, leveraging the husk operator for multi-GPU execution and introducing a worker-thread runtime environment for concatenation or reduction of the results on multiple devices. Next, the entire implementation is tested on Futhark's benchmark suite, investigating both the effect




(a) Bandwidth

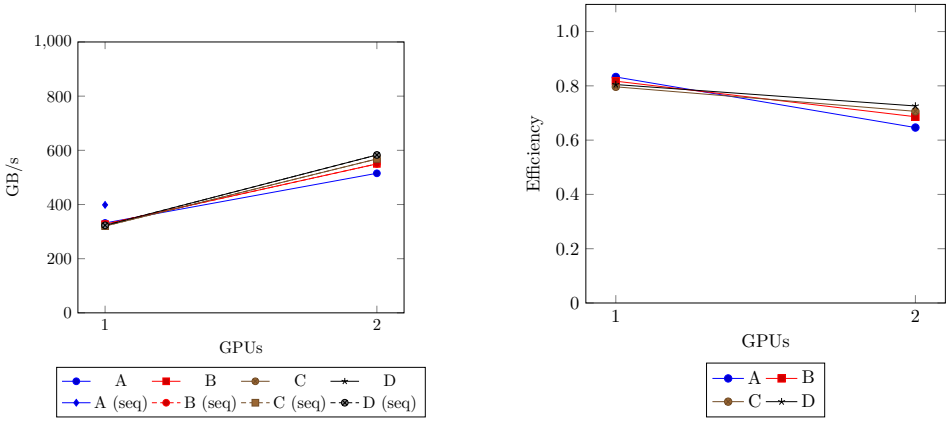
(b) Efficiency

Figure 7.11: Stencil on Snellius, 100 iterations

of the husk operator in single and double GPU scenarios. In the multi-GPU case, their results vary from near-linear speedups to limited improvements, depending on the amount of data that has to be transferred between the two GPUs. In single GPU scenarios the overhead of the husk operator is generally negligible, except for some outlier applications — ranging from an improvement of roughly 1.7× to significant slowdowns of 2.5×.

Accelerate [226] adds multi-GPU support by adding a fissioning program transformation during the compilation process, essentially converting intra-kernel data parallelism to inter-kernel task parallelism. Furthermore, a runtime multi-GPU scheduler is added, which decides what task is executed on which device. This scheduler resembles a typical approach, introducing a worker thread for each CUDA device. These worker threads are handling both kernel launches and copying dependencies to the GPU they represent. The authors evaluate their implementation against three benchmarks on a two-GPU system, where two of those three benchmarks are Matrix Multiplication and N-Body. For the N-Body, they achieve linear speedups and even a 2.4× speedup when the bodies fit in the cache of a single GPU. Their matrix multiplication benchmark achieves more modest speedups (around 1.3×), which the authors attribute to needing to communicate larger partial results and more expensive combination steps compared to their other benchmarks.

Another class of related work consists of more specialised transpilers and frameworks that tend to either implement a custom unified-memory protocol or extend CUDA's unified-memory driver [167, 129]. One such example is the framework SnuRHAC, which automatically and transparently distributes workloads to multiple GPUs across multiple nodes [129]. The authors extend CUDA unified memory with an additional Linux kernel module to achieve scaling beyond a single node of multiple GPUs. Within a node, kernel workloads are distributed by dividing the thread blocks in CUDA grids among multiple GPUs. Page faulting is also re-



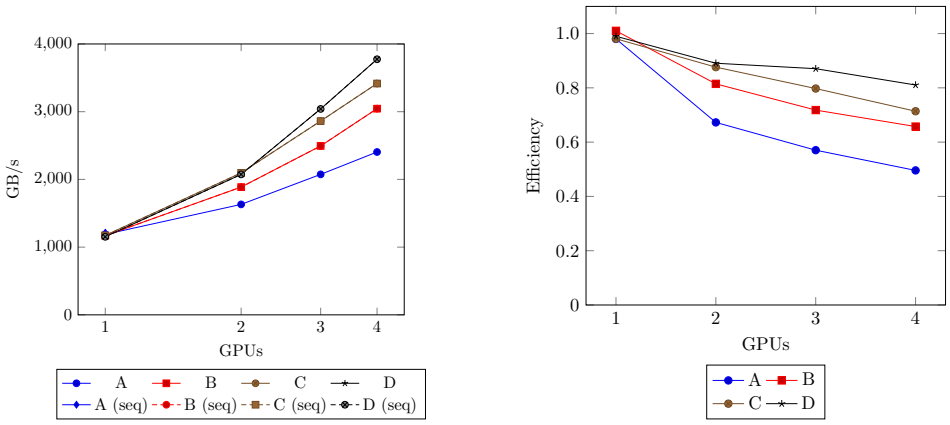

(a) Bandwidth (b) Efficiency

Figure 7.12: Stencil on Icis, 100 iterations

duced by implementing both static and dynamic page-prefetching techniques. The implementation also includes heuristics to decide whether to execute on a single GPU instead if the workload *e.g.* contains too many shared-write accesses.

Finally, we take a look at related work that conducts manual experimentation with CUDA unified memory. One study evaluated the effect of CUDA performance hints for both in-core and out-of-core single-GPU scenarios on two platforms across six different applications [52]. On one platform, they find that memory advice only lead to benefits in in-core scenarios, whereas on the other platform exclusively in out-of-core situations. When it comes to page prefetching, they find that it leads to significant improvements for one system, but no benefits on the other. Overall, the authors conclude that unified memory is a promising technology and point out that more research into optimal advice placement would benefit programmer productivity.

Another study investigates a manual multi-GPU implementation using unified memory [90]. The authors analyse both programmability and performance, focussing on the effect of performance hints. Their study considers two types of memory advice: access patterns and placement of memory, and experiments are also conducted with prefetching. The authors observe speedups ranging from 1.10× to 1.85× on their selected benchmarks. They find that programmability is generally much improved due to not needing to do manual allocations nor add explicit memory transfers, but at the same time performance hints are necessary to achieve good performance. When it comes to performance, unified memory can introduce communication optimisations due to only initiating transfers that are strictly necessary for a given computation.




(a) Bandwidth

(b) Efficiency

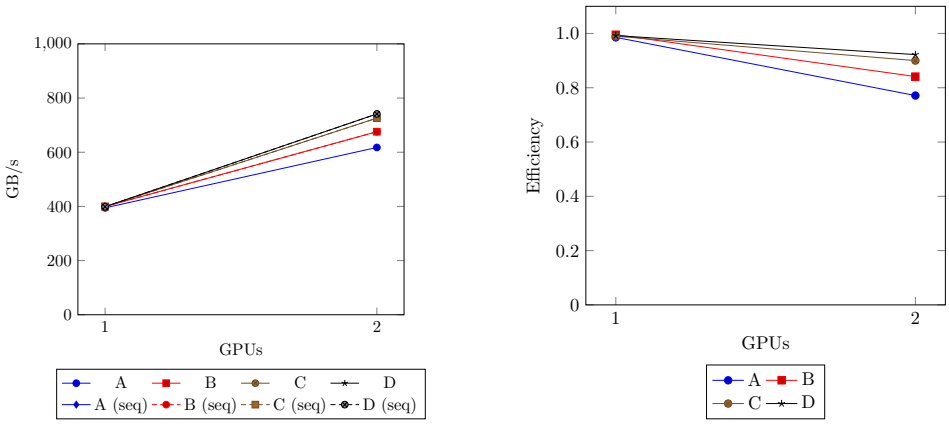
Figure 7.13: Stencil on Snellius, without allocation and deallocation

## 7.6 Conclusions and Future Work

We have implemented multi-GPU support for the language SaC extending its existing unified memory compilation target. This extension is light-weight as we can reuse the existing mechanism of specifying subsets. Inserting memory advice is crucial to obtain good performance on multiple GPUs. We can generate accurate advice due to the functional semantics of SaC.

Our benchmarks show promising results for both the N-Body and Matrix Multiply algorithms. With large enough problem sizes, we see efficiencies above 90% with respect to SaC's manually-allocated *cuda* target. For a nine-point stencil we can obtain efficiencies in excess of 80%, but only when we do not count allocation time. If the compiler cannot sufficiently reuse allocations, our approach yields efficiencies as low as 12%, even for large problem sizes.

Future work is needed to address the efficient support of reductions or fold on multiple GPUs. We use a basic device-wide synchronisation with expensive context switches between GPUs. This does not stop us from obtaining high efficiency on large problem sets, but fine-tuning this with event-based synchronisation may lead to performance benefits on smaller problem sizes.




(a) Bandwidth

(b) Efficiency

Figure 7.14: Stencil on Icis, without allocation and deallocation

# Part II

# Rank-polymorphic Algorithm Design

# Chapter 8

# Category Theory for Supercomputing

## 8.1 Introduction

Category theory is usually applied to other areas of abstract algebra, or theoretical computer science. In this chapter, we look at a more applied area: supercomputing (also called high-performance computing). It turns out that category theory is also useful in the very practical problem of implementing a fast algorithm for computing the Discrete Fourier Transform (DFT) on a supercomputer. This algorithm—called the Fast Fourier Transform (FFT)—implements the DFT in $O(n \log n)$ time for $n$ data points and is considered one of the top 10 algorithms of the 20th century [68].

Today, the 'super' in the word 'supercomputer' does not refer to a single processor that can compute very quickly, but rather to the number of processors. In fact, a supercomputer is a cluster of many connected processors, each with their own memory. Accessing data on a different processor requires explicit data movement and minimising this while maintaining load balance is the essence of making programs on supercomputers faster. We can reason about this by separating computation from data movement with a computational model called the *Bulk Synchronous Parallel* model [232], or *BSP* for short.

More specifically, the contribution of this chapter is Theorem 3, a recipe for taking the tensor product of certain BSP algorithms, formulated without category theory. The key idea behind the proof, however, is to use category theory as a tool to transfer knowledge from one area of mathematics to another, in this case from linear algebra to parallel algorithms. We can connect a certain class of BSP algorithms to constructs in linear algebra. This uncovers structure that is well-understood in the category Vect, and we use this theory to derive the core of our algorithm, namely distributing the tensor product over the direct sum. The result can then be translated back to the category Set, giving a fast BSP algorithm for higher-dimensional arrays. The application of Theorem 3 to the one-dimensional

algorithm of [120] gives the algorithm of Chapter 9, which shows improvements over the state-of-the-art. Though that chapter provides an elementary proof, the algorithm was derived using the theory from this chapter.

## 8.2 Background

### The Discrete Fourier Transform

The discrete Fourier Transform $F_n$ of length $n$ is a linear function $\mathbb{C}^n \to \mathbb{C}^n$. Writing $\omega_n$ for the $n$th root of unity $e^{-2\pi i/n}$, it is given explicitly by

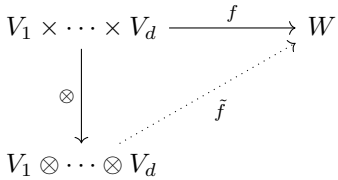
$$y_k := F_n(x)_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}.$$

There is also a multidimensional variant of the DFT that operates on $n_1 \times \cdots \times n_d$ arrays:

$$y_{k_1,\cdots,k_d} = \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} x_{j_1,\cdots,j_d} \omega_{n_1}^{j_1 k_1} \cdot \cdots \cdot \omega_{n_d}^{j_d k_d}. \tag{8.1}$$

We will call an $n_1 \times \cdots \times n_d$ array, an array of rank $d$ rather than dimension $d$ from now on in the tradition of the programming language APL [122]. For notational convenience, we will define $[n] := \{0, \cdots, n-1\}$ and abbreviate $(k_1, \cdots, k_d) \in [n_1] \times \cdots \times [n_d]$ as $k$.

The original motivation behind this work is to find an efficient parallel algorithm for the multidimensional DFT. To this end, it is important to understand what Eq. (8.1) means algebraically. The central concept behind generalizing the DFT from rank-1 to higher ranks is multilinearity: functions $f : V_1 \times \cdots \times V_d \to W$ that are linear in each argument. We may ask ourselves: what is the vector space that best captures multilinearity from $V_1 \times \cdots \times V_d$? The answer is the tensor product $V_1 \otimes \cdots \otimes V_d$, which together with a multilinear map $\otimes : V_1 \times \cdots \times V_d \to V_1 \otimes \cdots \otimes V_d$ can be characterised by the universal property that for any multilinear map $f : V_1 \times \cdots \times V_d \to W$, there exists a unique linear map $\tilde{f} : V_1 \otimes \cdots \otimes V_d \to W$ such that $\tilde{f} \circ \otimes = f$.




We can use this to turn a product of functions $(f_1, \cdots, f_d) : V_1 \times \cdots \times V_d \to W_1 \times \cdots \times W_d$ into a tensor product $f_1 \otimes \cdots \otimes f_d : V_1 \otimes \cdots \otimes V_d \to W_1 \otimes \cdots \otimes W_d$ by traversing the following commutative diagram from bottom left to top right to bottom right. Working out the tensor product of rank-1 DFTs yields the higher-ranked DFT of Eq. (8.1).

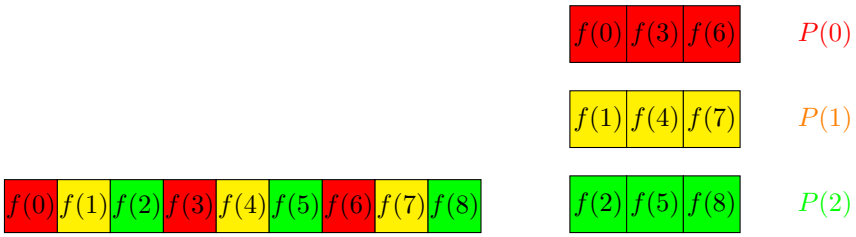
**for** $j = 0$ **to** $n$ **do**
  $X[j] = f(j)$

(a) Sequential initialisation

**for** $k = 0$ **to** $n/p$ **do**
  $X^{(s)}[k] = f(s + kp)$

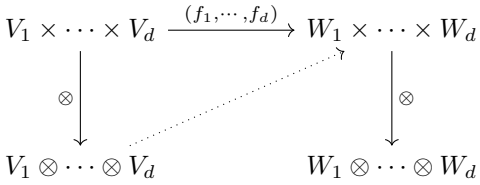
(b) Distributed initialisation on $P(s)$

Figure 8.1: Initialisation of $x_j = f(j)$ sequentially, and in parallel.




(a) Global view (b) Local view

Figure 8.2: Graphical representation of cyclically distributed initialisation over three processors, indicated in red, yellow, and green.




## Distributed Computing

In programming, there is usually little difference between data structures and the mathematical objects they represent. In the programming language FORTRAN, we would write `X(k)` for $x_k$ or `X(k1, k2)` for $x_{k_1,k_2}$. The programming language can then easily compute the address in memory from such an expression. This approach fails when the data structures are so large that they do not fit in the memory of a single machine. Programming a supercomputer is also called distributed-memory parallel computing because we have to distribute the data structure over different processors with their own memory. If we have $p$ processors, we will identify processors with an index $0 \leq s < p$. Each processor $P(s)$ has its local data structure $X^{(s)}$. These local data structures then implicitly define the global data structure $X$ through a correspondence called the *distribution*. Figure 8.1 gives an example, called the *cyclic distribution*. In this chapter, the **to** in algorithms is always exclusive. We can view $X$ as one data structure, called the *global view*, or as a collection of local data structures $(X^{(s)})_s$, called the *local view*, as illustrated in Fig. 8.2. The distribution can be interpreted as a bijection between the two views.

### The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model is often used, either implicitly or explicitly, for parallel computing. It comprises a parallel computer architecture, a class of parallel algorithms, and a function for charging costs to algorithms [232]. We employ the variant extensively used in [27] and focus on the algorithmic part.

*A BSP algorithm* consists of a sequence of supersteps. We have two kinds of supersteps. The first, a *computation* superstep, performs a sequence of operations on local data structures $X^{(s)}$. In the second, a *communication* superstep, each processor sends and receives a number of messages. At the end of a superstep, all processors synchronise as follows. Each processor checks whether it has finished all its tasks of that superstep. Processors wait until all others have finished. When this has happened, they all proceed to the next superstep. This form of synchronisation is called *bulk synchronisation*, hence the name of the model.

## 8.3 A BSP Algorithm for the Rank-1 DFT

To get a feel for BSP algorithms, we will derive a BSP algorithm for the four-step FFT framework, which can be found in a slightly different formulation in [233]. We use different variable names and notation to make it easier to understand the parallel algorithm. To that end, suppose $x$ is an array of length $n$, that $p \mid n$ and $p \mid \frac{n}{p}$ (or equivalently that $p^2 \mid n$), and that we want to calculate $y := F_n(x)$. We write $v(a : b : c)$ to mean the strided subarray of $v$ which starts at index $a$ and has stride $b$. The last variable $c$ is the length of $v$. If we write $v(a : b)$ we mean the subarray that starts at $a$ and ends at $b$ (exclusive). This lets us write the four-step framework as the sequential Algorithm 12.

**Algorithm 12** Sequential four-step FFT framework

**Input:** $x$ : array of length $n$, number $p$ such that $p^2 \mid n$.
**Output:** $y$ : array of length $n$, such that $y = F_n(x)$.

1: **for** $s \in [p]$ **do** ▷ Step 1
2: $\quad x^{(s)} = x(s : p : n)$;
3: $\quad x^{(s)} = F_{n/p}(x^{(s)})$;
4: **for** $s \in [p]$ **do** ▷ Step 2
5: $\quad$ **for** $k \in [n/p]$ **do**
6: $\quad\quad x_k^{(s)} = \omega_n^{ks} x_k^{(s)}$;
7: **for** $s \in [p]$ **do** ▷ Step 3
8: $\quad$ **for** $k \in [n/p]$ **do**
9: $\quad\quad y(k : \frac{n}{p} : n)_s = x_k^{(s)}$;
10: **for** $k \in [n/p]$ **do** ▷ Step 4
11: $\quad y(k : \frac{n}{p} : n) = F_p(y(k : \frac{n}{p} : n))$;

We use the parallelisation strategy of [120]. The notation we chose in Algorithm 12 already suggests using the cyclic distribution. This gives us the first two

steps of parallel Algorithm 13. A BSP algorithm in Single Program, Multiple Data (SPMD) style is written from the perspective of a processor, so the loop over $s$ is removed and $s$ becomes the processor index.

**Algorithm 13** Parallel four-step FFT framework for $P(s)$

**Input:** $x$ : array of length $n$, distr$(x)$ = cyclic over $p$ processors such that $p^2 \mid n$.
**Output:** $y$ : array of length $n$, distr$(y)$ = cyclic, such that $y = F_n(x)$.

1: $x^{(s)} = F_{n/p}(x^{(s)})$; ▷ Step 1
2: **for** $k \in [n/p]$ **do** ▷ Step 2
3: $\quad x_k^{(s)} = \omega_n^{ks} x_k^{(s)}$;
4: **for** $k \in [p]$ **do** ▷ Step 3
5: $\quad$ Put $x^{(s)}(k : p : \frac{n}{p})$ in $P(k)$ as $y^{(k)}(s\frac{n}{p^2} : (s+1)\frac{n}{p^2})$;
6: **for** $t \in [n/p^2]$ **do** ▷ Step 4
7: $\quad y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}) = F_p(y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}))$;

In the BSP framework we view Step 1 and Step 2 together as one computation superstep, Step 3 as a communication superstep, and Step 4 as a computation superstep as well. Steps 3 and 4 are derived as follows.

To parallelize Step 3, we decompose $k$ as $k = k' + cp, \quad 0 \le k' < p, \quad 0 \le c < n/p^2$ so we can consider the $k$th element of $x^{(s)}$ to be the $c$th element of the strided subarray $x^{(s)}(k' : p : n/p)$. The following index chase shows that this strided subarray becomes the $s$th contiguous subarray on $P(k')$.

$$\begin{aligned} y(k : n/p : n)_s &= y_{k+sn/p} \\ &= y_{k'+(sn/p^2+c)p} \\ &= y^{(k')}_{sn/p^2+c} \\ &= y^{(k')}(sn/p^2 : (s+1)n/p^2)_c. \end{aligned}$$

Finally, we drop the prime from $k'$ to obtain Step 3 of the parallel algorithm.

To parallelize Step 4, we first decompose $k$ as $k = tp + s, \quad 0 \le s < p, \quad 0 \le t < n/p^2$. We then use our assumption that $p^2 \mid n$ to rewrite the $c$th index of $y(k : n/p : n)$ as $k + c\frac{n}{p} = s + (t + cn/p^2)p$. We conclude that the $c$th element of $y(k : n/p : n)$ is the $(t + cn/p^2)$th element of $y^{(s)}$, which means it is the $c$th element of $y^{(s)}(t : n/p^2 : n/p)$.

As a result, we have obtained a complete BSP algorithm for the fast computation of a rank-1 DFT.

## 8.4 Linear BSP Algorithms

To apply theory from linear algebra, we must first translate the structure of a BSP algorithm to Vect. We restrict the class of BSP algorithms to those that compute linear functions using supersteps that respect linearity in a way that we define more precisely in Definition 1. The BSP model does not explicitly mention

distributions because they do not incur computation or data movement, but they are algebraically important. We discuss distributions, computation supersteps, and communication supersteps.

## Arrays as Vectors

A computer program for a linear function $f$, is not bound by linear structure. So writing $U$ for the forgetful functor, we compute $Uf$.

The forgetful functor $U : \mathrm{Vect} \to \mathrm{Set}$ is accompanied by a free functor $V : \mathrm{Set} \to \mathrm{Vect}$ that adds structure. This functor is used whenever we work with arrays: an array $X$ corresponds to the vector $\sum_{j\in[n]} X[j]b_j$, where $(b_j)_{j\in[n]}$ is a basis of $\mathbb{C}^n$. The free functor applied to a set $B$ returns a vector space $VB$ consisting of all formal linear combinations of elements in $B$ (so $B$ is a basis of $VB$). This means that we can interpret the set of arrays over an index set $I$ as $UVI$. We can use this view to explain why we represent the tensor product of DFTs as an operation on $n_1 \times \cdots \times n_d$ arrays in Eq. (8.1). We can form a basis of the tensor product $VB_1 \otimes \cdots \otimes VB_d$ by taking the product of bases $B_1 \times \cdots \times B_d$, so $VB_1 \otimes \cdots \otimes VB_d \cong V(B_1 \times \cdots \times B_d)$. Given rank-1 index sets of the form $I = [n]$, this gives a $d$-dimensional index set of the form $[n_1] \times \cdots \times [n_d]$, which indexes a rank-$d$ array.

## Distributions

In the BSP setting we have a collection of index sets $(I^{(s)})_{s\in[p]}$ for the processors, rather than one global index set $I$. A distribution is then a collection of functions $\phi^s : I^{(s)} \to I$ that links these through $X^{(s)}[k] = X[\phi^s(k)]$. In other words, we have a disjoint union $\coprod_{s\in[p]} I^{(s)}$ and an induced function $\phi = [\phi^s \mid s \in [p]] : \coprod_{s\in[p]} I^{(s)} \to I$ that is bijective. The free functor turns this into the bijection between the local and global view of Fig. 8.2.

Viewing distributions as bijections between index sets allows for an algebraic interpretation of parallelism: the disjoint union is the coproduct in Set, and as $V \dashv U$, the free functor preserves this colimit and maps it to the coproduct in Vect, the direct sum. The direct sum is also the product in Vect, which is preserved by the forgetful functor. This gives the commutative diagram of Fig. 8.3, where we use the square braces for the unique morphism out of coproducts in both Set and Vect.

The left side of Fig. 8.3, shows that a distributed array is really just an array with index set $\coprod_{s\in[p]} I^{(s)}$, and that the distribution is an isomorphism between this array and $X$. In Fortran, this is how we think about distributed computing when using coarrays: we can declare a distributed array of rank two as `X(:,:)[*]`, and then refer to $X^{(s)}$ by `X(:,:)[s]`. The top right side, $\prod_{s\in[p]} UVI^{(s)}$, views a distributed array as a collection of local data structures. This is how we usually think about distributed computing in the BSP model. The top middle part exposes the algebraic meaning of parallelism: the direct sum.

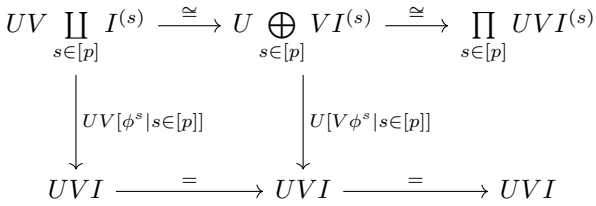



Figure 8.3: Distributions correspond to the direct sum in Vect. Local view at the top, global view at the bottom.

## Computation Superstep

A computation superstep is a collection of functions between these data structures, so a product. If these functions are linear, the product is preserved under the forgetful functor, so they correspond to a direct sum as illustrated in Fig. 8.4. Note that this holds for Algorithm 13 as $F_{n/p}$, $F_p$ and $x_k^{(s)} \mapsto \omega_n^{ks} x_k^{(s)}$ are all linear.

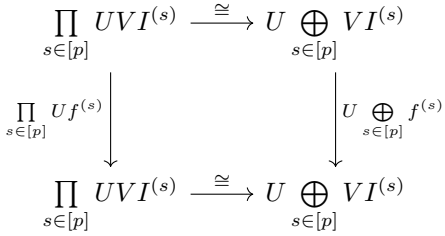



Figure 8.4: A BSP computation superstep of linear functions

## Communication Superstep

The communication superstep of Algorithm 13 permutes the local arrays, meaning it can be described as an isomorphism on the index set $\coprod_{s\in[p]} I^{(s)}$. In our experience, many communication supersteps in parallel algorithms can be described this way, so we restrict ourselves to communication supersteps of this type.

As functions out of a coproduct are determined uniquely by their components, we can write such an isomorphism as $r = [r_s \mid s \in [p]]$. Concretely, we have $r_s(k) = (k \bmod p, sn/p^2 + k \operatorname{div} p)$ in Step 3 of Algorithm 13 because we put the $k$th element of $P(s)$ in the $(sn/p^2 + k \operatorname{div} p)$th position on $P(k \bmod p)$.

A permutation on basis elements is lifted to an isomorphism between vector spaces by the free functor, so we consider isomorphisms as in Fig. 8.5.

$$\begin{array}{c} UV \coprod\limits_{s\in[p]} I^{(s)} \\ \Big\downarrow {\scriptstyle UV[r_s|s\in[p]]} \\ UV \coprod\limits_{s\in[p]} I^{(s)} \end{array}$$

Figure 8.5: A BSP communication superstep that redistributes data is a bijection on the index set under the free functor

### Definition

To conclude, the type of BSP algorithm that we consider factorises a linear function $f$ in direct sums of functions and index permutations under the free functor. We then forget the linear structure by using the forgetful functor. More precisely:

**Definition 1** (Linear BSP algorithm)**.** Interpret arrays $X$ and $Y$ over index sets $I_x$, $I_y$ as $Ux$, $Uy$ for vectors $x \in VI_x$, $y \in VI_y$. A *linear BSP algorithm* has an array $X$ as input, an array $Y$ as output, and computes $y = f(x)$ for a linear function $f$ such that the following conditions hold:

1. all computation supersteps are of the form $Y^{(s)} = Uf^{(s)}(X^{(s)})$ for linear functions $f^{(s)}$;
2. all communication supersteps are of the form $UVr$ where $r$ permutes the index set $\coprod_{s\in[p]} I^{(s)}$.

## 8.5 The Tensor Product of Linear BSP Algorithms

We can now state Theorem 3, which shows that if we have linear BSP algorithms for functions $f_1, \cdots, f_d$, we can straightforwardly derive a BSP algorithm of the same structure for their tensor product $f_1 \otimes \cdots \otimes f_d$.

**Theorem 3** (The tensor product of linear BSP algorithms)**.** *Linear BSP algorithms $\mathcal{A}_1, \cdots \mathcal{A}_d$ for $f_1, \cdots, f_d$ with the same number and structure of supersteps can be combined into a linear BSP algorithm for $f_1 \otimes \cdots \otimes f_d$ by combining the supersteps and distributions as follows.*

- *If $\mathcal{A}_l$ uses $p_l$ processors indexed between 0 and $p_l$, the higher-dimensional algorithm uses a $d$-dimensional processor grid $p := [p_1] \times \cdots \times [p_d]$.*
- *If $\mathcal{A}_l$ has distribution*
$$[\phi^{(s_l)} \mid s_l \in [p_l]],$$
*the higher-dimensional algorithm has distribution*
$$[\phi^{s_1} \times \cdots \times \phi^{s_d} \mid s \in p].$$

- *Computation supersteps that compute* $Ug_1, \cdots, Ug_d$ *are combined into a computation superstep that computes* $U(g_1 \otimes \cdots \otimes g_d)$.
- *Communication supersteps that perform a permutation*

$$Z^{(\psi_l(s_l,k_l))}[\rho_l(s_l,k_l)] = X^{(s_l)}[k_l]$$

*are combined into a communication superstep that performs the permutation*

$$Z^{((\psi_1 \times \cdots \times \psi_d)(s,k))}[(\rho_1 \times \cdots \times \rho_d)(s,k)] = X^{(s)}[k].$$

(Note that we can always add identity functions to make the number and structure of supersteps match.)

The proof relies on two facts. First, the tensor **product** and direct **sum** distribute over each other in the same way the product and sum of numbers do:

$$\left(\bigoplus_{s_1 \in [p_1]} VI^{(s_1)}\right) \otimes \cdots \otimes \left(\bigoplus_{s_d \in [p_d]} VI^{(s_d)}\right) \cong \bigoplus_{s \in p} (VI^{(s_1)} \otimes \cdots \otimes VI^{(s_d)}). \quad (8.2)$$

This isomorphism is a natural transformation and allows us to pull the direct sum (encoding parallelism) out of the tensor product. Second, the tensor product is functorial, so we can study the distribution, computation superstep, and communication superstep in isolation.

## Distributions

Equation (8.2) is the intuition behind the tensor product turning a collection of distributed rank-1 arrays into a distribution of a rank-$d$ array, with also a rank-$d$ processor grid. We work this out in more detail in Fig. 8.6, where we also use that the tensor product is the image of the Cartesian product in Set under the free functor. We can tell that the $\phi_{s_l}$ are combined dimension-wise by comparing the right side of Fig. 8.6 with the left side of Fig. 8.3.

$$\begin{array}{ccccccc}
\bigotimes_{l=1}^{d} \bigoplus_{s_l \in [p_l]} VI^{(s_l)} & \xrightarrow{\cong} & \bigoplus_{s \in p} \bigotimes_{l=1}^{d} VI^{(s_l)} & \xrightarrow{\cong} & \bigoplus_{s \in [p]} V \prod_{l=1}^{d} I_l^{(s)} & \xrightarrow{\cong} & V \coprod_{s=0}^{p-1} \prod_{l=1}^{d} I_l^{(s)} \\
\downarrow{\scriptstyle \bigotimes_{l=1}^{d} [V\phi^{s_l}]} & & \downarrow{\scriptstyle [\bigotimes_{l=1}^{d} V\phi^{s_l}]} & & \downarrow{\scriptstyle [V \prod_{l=1}^{d} \phi^{s_l}]} & & \downarrow{\scriptstyle V[\prod_{l=1}^{d} \phi^{s_l}]} \\
\bigotimes_{l=1}^{d} VI_l & \xrightarrow{=} & \bigotimes_{l=1}^{d} VI_l & \xrightarrow{\cong} & V \prod_{l=1}^{d} I_l & \xrightarrow{=} & V \prod_{l=1}^{d} I_l
\end{array}$$

Figure 8.6: The tensor product of a distribution is again a distribution, where we take the Cartesian product of index sets and distribution functions

## Computation Superstep

Equation (8.2) directly gives the diagram of Fig. 8.7. We conclude that we can simply apply the tensor product locally.

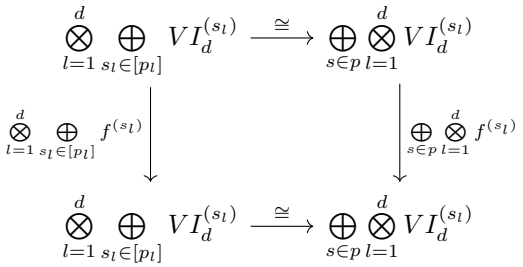


Figure 8.7: The tensor product of a BSP computation superstep of linear functions

## Communication Superstep

The right side of Figure 8.8 shows that the tensor product of a communication superstep is the product of the underlying permutations composed with an isomorphism.

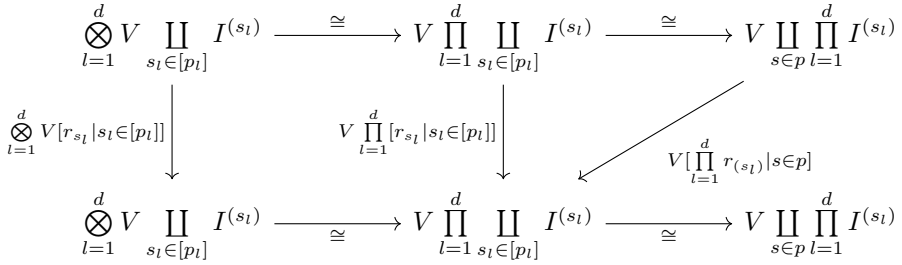


Figure 8.8: The tensor product of a redistribution is done dimension-wise

To show how this corresponds to Theorem 3, we chase through the right side of the diagram. We write $r_{s_l}(k) = (\psi_l(s_l, k), \rho_l(s_l, k))$, which is interpreted as: Put $X^{(s_l)}[k]$ into $P(\psi_l(s_l, k))$ at local index $\rho_l(s_l, k)$. Starting at the top right corner, and ignoring the free functor $V$, we get

$$
\begin{aligned}
&((s_1, \cdots, s_d), (k_1, \cdots, k_d)) \mapsto \\
&(r_{s_1}(k_1), \cdots, (r_{s_d}(k_d)) = \\
&((\psi_1(s_1, k_1), \rho_1(s_1, k_1)), \cdots, (\psi_d(s_d, k_d), \rho_d(s_d, k_d)) \mapsto \\
&((\psi_1(s_1, k_1), \cdots, \psi_d(s_d, k_d)), (\rho_1(s_1, k_1), \cdots, \rho_d(s_d, k_d)).
\end{aligned}
$$

## 8.6 Applications

### Discrete Fourier Transform

Theorem 3 can be used to directly derive Algorithm 14 from Algorithm 13, without the tedious proof of [142].

**Algorithm 14** Parallel four-step FFT framework for processor $P(s) = P(s_1, \cdots, s_d)$, rank-$d$

**Input:** $X$ : array of size $n_1 \times \cdots \times n_d$, distr$(X)$ = rank-$d$ cyclic over $p_1 \times \cdots \times p_d$ processors such that $p_l^2 | n_l$, for $l = 1, \cdots, d$.

**Output:** $Y$ : array of size $n_1 \times \cdots \times n_d$, distr$(Y)$ = rank-$d$ cyclic, such that $Y = (F_{n_1} \otimes \cdots \otimes F_{n_d})(X)$.

1: $X^{(s)} := (F_{n_1/p_1} \otimes \cdots \otimes F_{n_d/p_d})(X^{(s)})$; ▷ Superstep 0
2: **for** $k \in [n_1/p_1] \times \cdots \times [n_d/p_d]$ **do**
3: $\quad X^{(s)}[k] := (\prod_{l=1}^{d} \omega_{n_l}^{k_l s_l}) X^{(s)}[k]$;
4: **for** $k \in [p_1] \times \cdots \times [p_d]$ **do** ▷ Superstep 1
5: $\quad$ Put $X^{(s)}(k : p : \frac{n}{p})$ in $P(k)$ as $Y^{(k)}[s\frac{n}{p^2} : (s+1)\frac{n}{p^2}]$;
6: **for** $t \in [n_1/p_1^2] \times \cdots \times [n_d/p_d^2]$ **do** ▷ Superstep 2
7: $\quad Y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}) := (F_{p_1} \otimes \cdots \otimes F_{p_d}) \left( Y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}) \right)$;

### Discrete Cosine Transform

We can also apply Theorem 3 to the Discrete Cosine Transform (DCT-II), see e.g. [233, 197]:

$$y_k = \sum_{j=0}^{n-1} x_j \cos \frac{(2j+1)k\pi}{2n}. \tag{8.3}$$

The most efficient sequential rank-1 DCT-II algorithms pack the $n$ real input points into $n/2$ complex data points, apply the DFT, and then extract the result. Our approach is not applicable to these algorithms because the extraction is linear with respect to $\mathbb{R}$, but not $\mathbb{C}$. This mix of ground fields breaks the theory we have developed. We discuss this further in Section 8.9. Instead we use a less efficient algorithm that views the DCT-II as a complex function, that we happen to apply to real input only.

We can implement the DCT-II by extending the signal, applying a DFT, and then extracting the result [164]. Given an input $x$ of length $n$, we define $w$ of length $2n$ as

$$w_j = \begin{cases} x_j & 0 \leq j < n \\ x_{2n-1-j} & n \leq j < 2n. \end{cases}$$

After transforming $z = DFT(w)$, the DCT-II $y$ can be extracted by

$$y_k = \frac{1}{2}\omega_{2n}^{k/2} z_k, \quad 0 \leq k < n.$$

**Linear BSP Algorithm**

To derive a linear BSP algorithm, we first decompose this algorithm into functions. We write $r$ for the reversal function $x_j \mapsto y_{n-1-j}$, $\langle f, g \rangle$ for the function $V \to W_1 \oplus W_2$ induced by $f : V \to W_1$, $g : V \to W_2$, $\pi_1$ for the projection to the first component, and $\mathbb{C}^n \oplus \mathbb{C}^n \cong \mathbb{C}^{2n}$ for putting the bases after each other. This gives the following decomposition.

$$\mathbb{C}^n \xrightarrow{\langle \mathrm{id}, \mathrm{id} \rangle} \mathbb{C}^n \oplus \mathbb{C}^n \xrightarrow{\mathrm{id} \oplus r} \mathbb{C}^n \oplus \mathbb{C}^n \xrightarrow{\cong} \mathbb{C}^{2n} \xrightarrow{F_{2n}}$$

$$\mathbb{C}^{2n} \xrightarrow{\cong} \mathbb{C}^n \oplus \mathbb{C}^n \xrightarrow{\pi_1} \mathbb{C}^n \xrightarrow{\cdot \frac{1}{2}\omega_{2n}^{k/2}} \mathbb{C}^n$$

To ensure a cyclic distribution for the DFT, we take a cyclic distribution of $\mathbb{C}^n$. Local index $k$ on $P(s)$ corresponds to global index $s + kp$. If we embed this in the second component of $\mathbb{C}^n \oplus \mathbb{C}^n \cong \mathbb{C}^{2n}$, we get global index $n+s+kp = s+(n/p+k)p$, which shows it is the $k$th local index in the second component of $\mathbb{C}^{n/p} \oplus \mathbb{C}^{n/p}$ on $P(s)$. So the duplication $\langle \mathrm{id}, \mathrm{id} \rangle$ becomes a local duplication under the cyclic distribution. As this is a local linear computation, it fits in our framework as a computation superstep.

The function $\mathrm{id} \oplus r$ is not local under the cyclic distribution, so we model it as a communication superstep. Local index $k$ on $P(s)$ has global index $s + kp$, and $r(s + kp) = n - 1 - (s + kp) = p - 1 - s + (n/p - 1 - k)p$. So the reverse is stored on $P(p - 1 - s)$ in local index $n/p - 1 - k$. To deal with the embedding $\mathbb{C}^{n/p} \oplus \mathbb{C}^{n/p} \cong \mathbb{C}^{2n/p}$, we first subtract $n/p$ from $k$ to obtain $k' = k - n/p$, then reverse to give the index $n/p - k' - 1$, and finally add $n/p$ back to the result, which gives local index $3n/p - 1 - k$. In the notation of Theorem 3, this gives the following communication step.

$$\rho(s, k) = \begin{cases} k & 0 \leq k < n/p \\ 3n/p - 1 - k & n/p \leq k < 2n/p \end{cases}$$

$$\psi(s, k) = \begin{cases} s & 0 \leq k < n/p \\ p - 1 - s & n/p \leq k < 2n/p \end{cases}$$

**Tensor Product**

To apply Theorem 3, we must form the tensor product of computation supersteps

$$\mathbb{C}^{n/p} \xrightarrow{\langle \mathrm{id},\mathrm{id}\rangle} \mathbb{C}^{n/p} \oplus \mathbb{C}^{n/p} \cong \mathbb{C}^{2n/p}$$

and

$$\mathbb{C}^{2n/p} \cong \mathbb{C}^{n/p} \oplus \mathbb{C}^{n/p} \xrightarrow{\pi_1} \mathbb{C}^{n/p} \xrightarrow{\cdot \frac{1}{2}\omega_{2n_l}^{k_l/2}} \mathbb{C}^{n/p}.$$

The tensor product respects the product in the obvious way, e.g.

$$\bigotimes_{l=1}^{d} \langle \mathrm{id}_{c_l} \mid c_l \in [2]\rangle \cong \langle \mathrm{id}_{c_1} \otimes \cdots \otimes \mathrm{id}_{c_d} \mid c \in [2] \times \cdots \times [2]\rangle.$$

The projections are similarly combined dimension-wise. To compose with $\mathbb{C}^{2n/p} \cong \mathbb{C}^{n/p} \oplus \mathbb{C}^{n/p}$, we can add $cn/p$ to the local index. The pointwise product is multiplied across dimensions. This yields Algorithm 15.

**Algorithm 15** Parallel DCT-II for processor $P(s) = P(s_1, \cdots, s_d)$, rank-$d$

**Input:** $X$ : array of size $n_1 \times \cdots \times n_d$, $\mathrm{distr}(X) =$ rank-$d$ cyclic over $p_1 \times \cdots \times p_d$ processors such that $p_l^2 | 2n_l$, for $l = 1, \cdots, d$.

**Output:** $Y$ : array of size $n_1 \times \cdots \times n_d$, $\mathrm{distr}(Y) =$ rank-$d$ cyclic, such that $Y = (C_{n_1} \otimes \cdots \otimes C_{n_d})(X)$.

1: **for** $c \in [2] \times \cdots \times [2]$ **do**
2: **for** $k \in [n_1/p_1] \times \cdots \times [n_d/p_d]$ **do**
3: $W^{(s)}[cn/p + k] = X^{(s)}[k]$;
4: **for** $k \in [2n_1/p_1] \times \cdots \times [2n_d/p_d]$ **do**
5: $W'^{((\psi_1 \times \cdots \times \psi_d)(s,k))}[(\rho_1 \times \cdots \times \rho_d)(k)] = W^{(s)}[k]$;
6: Compute $Z = DFT(W')$ with the parallel four-step FFT framework;
7: **for** $k \in [n_1/p_1] \times \cdots \times [n_d/p_d]$ **do**
8: $Y^{(s)}[k] = \prod_{l=1}^{d} \omega_{2n_l}^{k_l/2} Z^{(s)}[k]$;

## 8.7 Related Work

Johnson [128] has paved the way towards a common framework for many DFT algorithms by using matrix decomposition. This is also the fundamental unifying approach in the book by Van Loan [233]. The main difference with our work is that a matrix representation requires bases to be ordered linearly, losing the higher-ranked structure of the DFT. This has led to a more mechanical approach, deriving matrix-decompositions that are technically correct, but can be conceptually misleading. For example, Johnson uses $F_{n/p} \otimes I_p$ for a cyclic distribution and $I_p \otimes F_{n/p}$ for a block distribution, while we would consider both operations as $\bigoplus_{s\in[p]} F_{n/p}$ with a different isomorphism $\bigoplus_{s\in[p]} \mathbb{C}^{n/p} \cong \mathbb{C}^n$. The multiple $F_{n/p}$ in

Algorithm 12 arise from recursively applying the Discrete Fourier Transform on subarrays, not from multilinearity, which is why we believe the direct sum is a better choice than the $\otimes$-notation.

Designing and implementing algorithms often goes hand-in-hand. We speculate that previous work has sacrificed the higher-ranked structure because arrays of arbitrary rank are challenging to work with in programming languages commonly used for supercomputing, such as Fortran, C, or C++. This is unfortunate, as the structure of Algorithm 14 does not depend on rank, so we would prefer to have one subroutine that can take an array of any rank as argument. This is called *rank-polymorphism* and is supported by languages like APL [122] and Single-assignment C [208]. We show that rank-polymorphism can be used to describe tensor products of functions, but it has also found applications in controlling concurrency and data movement [209].

## 8.8 Conclusion

We introduced the notion of linear BSP algorithms: a class of parallel algorithms that respects linearity and that follows the Bulk Synchronous Parallel model for designing algorithms. This model distributes data structures over multiple processors with their own memory. Computation is done on these local data structures, and separated from communication by global barrier synchronisation. The split of computation and communication makes it possible to view algorithms as a form of function decomposition into distributions, computation supersteps, and communication supersteps. Investigating what the distributions, computation, and communication look like in the category Vect led to the main result of this chapter: that linear BSP algorithms for a collection of functions $f_1, \cdots, f_d$ give a linear BSP algorithm of the same structure for their tensor product $f_1 \otimes \cdots \otimes f_d$ (Theorem 3).

## 8.9 Future Work

Future work could investigate what parallelism looks like in other categories, and if it interacts with universal constructions in interesting ways. For example, there are many algorithms that use complex conjugation, which is not a linear function as $\overline{(\lambda \cdot z)} = \overline{\lambda} \cdot \overline{z}$. The concept of conjugation is captured and generalized in so-called $*$-algebras. Functions that respect addition, but conjugate scalar multiplication are called *anti-linear*. These functions are ubiquitous in the study of Hilbert spaces, making related categories of practical interest.

More efficient parallel algorithms for the DCT-II exist, based on using the DFT and functions that are linear with respect to $\mathbb{R}$, but not $\mathbb{C}$ [181, 119]. This mix of ground fields keeps us from applying Theorem 3, as we deal with $\otimes_{\mathbb{R}}$ and $\otimes_{\mathbb{C}}$, which live in different categories. Future work could find a more efficient algorithm by parallelising Feig and Winograd's algorithm [73] using the BSP model. This algorithm uses only real numbers, making it suitable for our approach.

Other interesting applications that connect algebraic structures with applications can be found in cryptography and topological data analysis.

The tensor product also describes the combination of Hadamard gates, making it fundamental in quantum computing [184]. As the performance of quantum computers relies on parallelism, it may also be a good candidate for the higher level of abstraction that category theory can provide.

# Chapter 9

# Minimizing Communication in the Multidimensional FFT

## 9.1 Introduction

The one-dimensional discrete Fourier transform (1D DFT) of length $n$ is a function $F_n$ that takes an array $x$ of $n$ complex numbers, and returns an array $y = F_n(x)$ of $n$ complex numbers. We write $\omega_n$ for the $n$th root of unity $e^{-2\pi i/n}$ and $[n]$ for $\{0, \ldots, n-1\}$. The DFT is given by

$$y_k = \sum_{j \in [n]} x_j \omega_n^{jk}, \quad \text{for } k \in [n]. \tag{9.1}$$

The fast Fourier transform [56] is a fast implementation of the DFT that can perform the computation in $\mathcal{O}(n \log n)$ time instead of the $\mathcal{O}(n^2)$ time of a straightforward implementation of Eqn 9.1; see [233] for a detailed exposition.

We also have a multidimensional variant of the DFT, which takes $n_1 \times \cdots \times n_d$ multidimensional arrays of complex numbers as both input and output, and is given by

$$Y[k_1, \ldots, k_d] = \sum_{j_1 \in [n_1]} \cdots \sum_{j_d \in [n_d]} X[j_1, \ldots, j_d] \omega_{n_1}^{j_1 k_1} \cdots \omega_{n_d}^{j_d k_d}, \tag{9.2}$$
$$\text{for } (k_1, \ldots, k_d) \in [n_1] \times \cdots \times [n_d].$$

We find it convenient in our notations to number the dimensions of the FFT starting at 1, but to have all other numberings start at 0. Without loss of generality, we assume that $n_1 \geq n_2 \geq \cdots \geq n_d$. We write $N = n_1 \cdots n_d$ for the total number of array elements.

The multidimensional FFT is the main computational kernel in many applications. An example is the solution of the time-dependent Schrödinger equation by wave packet propagation [147, 153], where the FFT is used in a spectral method

to compute the kinetic-energy operation efficiently, and where the dimension increases rapidly with the number of atoms (three per additional atom), see [36] for a 6D calculation. Other examples are classical molecular dynamics (MD), where a 3D FFT is used to compute long-range interactions efficiently based on Ewald sums, for instance in the LAMMPS package [193], and 3D computed tomography [199]. These are just a few examples; there are numerous other applications ranging from image processing, fluid flow simulation, to weather and climate prediction.

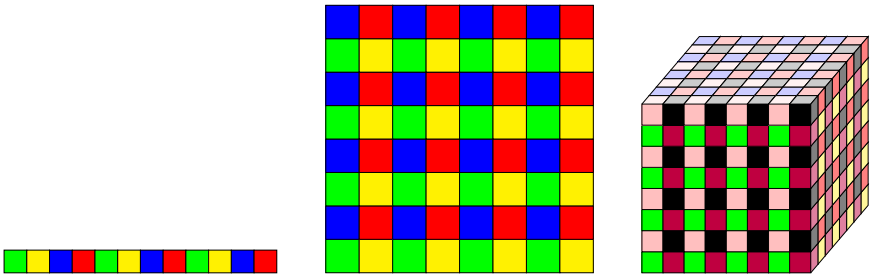
## Distributions

Informally, a data distribution, or *distribution* is a specification on how to store a data structure divided over a set of processors. Suppose we have a global data structure $X$ indexed by some set $J$, e.g. $J = [n]$ if $X$ is a 1D array of size $n$, or $J = [n_1] \times \cdots \times [n_d]$ for $X$ a $d$-dimensional array of size $n_l$ in dimension $l$. Let $\{P(s) \mid s \in S\}$ be a set of processors. We denote the local data structure on $P(s)$ by $X^{(s)}$ and assume it is indexed by a set $J_s$. A distribution is then more formally a bijection $\phi : \sqcup_{s \in S} J_s \to J$ from the disjoint union of local index sets to the global index set, where $\phi$ is defined by $X^{(s)}[k]$ corresponding to $X[\phi(s,k)]$.

The *cyclic distribution* of a 1D array of length $n$ over a set of processors indexed by $[p]$, is given by $\phi(s,k) = s + kp$. We generalize this to a $d$-dimensional cyclic distribution over a set of processors indexed by $[p_1] \times \cdots \times [p_d]$ for $d$-dimensional arrays. The distribution is then given by $\phi((s_1,\ldots,s_d),(k_1,\ldots,k_d)) = (s_1 + k_1p_1,\ldots,s_d + k_dp_d)$. When the dimension is clear, we will abbreviate this to $\phi(s,k) = s + kp$. The cyclic distribution has been shown to be advantageous for the 1D parallel FFT [120], because it can be used as both the input and output distribution for the FFT while requiring only one data redistribution during the computation; see also [27] for an extensive discussion. The cyclic distribution can be used for up to $p = \sqrt{n}$ processors. It has been shown to achieve state-of-the-art performance in a 1D implementation that uses FFTW for sequential FFTs and MulticoreBSP for C for communication [245]. The 1D, 2D, and 3D cyclic distributions for 1D, 2D, and 3D arrays, respectively, are illustrated in 9.1.

The *slab distribution* (or *1D distribution*) distributes a multidimensional array by blocks along one dimension. If we have $p$ processors and distribute along the first dimension, the distribution is given by $\phi(s,(k_1,\ldots,k_d)) = (k_1 + s \cdot \frac{n}{p}, k_2,\ldots,k_d)$. Distribution along another dimension can be done analogously. This is illustrated in 9.2.

The *pencil distribution* (or *2D distribution*) is similar to the slab distribution, but it distributes along two dimensions. If the processor indexing set is $[p_1] \times [p_2]$ with $p = p_1p_2$, the pencil distribution is given by $\phi((s_1,s_2),(k_1,\ldots,k_d)) = (k_1 + s_1 \cdot \frac{n_1}{p_1}, k_2 + s_2 \cdot \frac{n_2}{p_2}, k_3,\ldots,k_d)$. This is illustrated in 9.3. The pencil distribution can easily be generalized to a higher-dimensional distribution.



(a) An array of length 8 distributed over 4 processors in a cyclic distribution.

(b) An $8 \times 8$ array distributed over 4 processors in a $2 \times 2$-cyclic distribution.

(c) An $8 \times 8 \times 8$ array distributed over 8 processors in a $2 \times 2 \times 2$-cyclic distribution.

Figure 9.1: Cyclic distribution in several dimensions, indicated by colors.

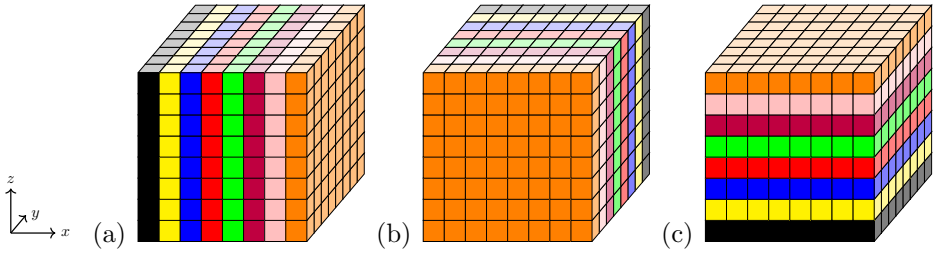



Figure 9.2: An $8 \times 8 \times 8$ array distributed over 8 processors in a slab distribution along different dimensions. (a) A slab distribution along the $x$-direction; (b) along the $y$-direction; (c) along the $z$-direction.

## Related work

Many state-of-the-art parallel FFT libraries, including the packages FFTW [81], PFFT [191], and HeFFTe [13] that we will use in our comparisons, exploit that the sum describing the FFT factorizes as follows

$$\sum_{j_1 \in [n_1]} \cdots \sum_{j_d \in [n_d]} X[j_1, \ldots, j_d] \omega_{n_1}^{j_1 k_1} \cdots \omega_{n_d}^{j_d k_d} = \sum_{j_1 \in [n_1]} \left( \sum_{j_2 \in [n_2]} \cdots \sum_{j_d \in [n_d]} X[j_1, \ldots, j_d] \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d} \right) \cdot \omega_{n_1}^{j_1 k_1}. \quad (9.3)$$

We see that the $d$-dimensional Fourier transform can be calculated in two steps: first, we apply a $(d-1)$-dimensional FFT to $X[j_1, *, \ldots, *]$ for all $j_1$, calling the result $Y$ (usually this is done in place, so with the same data structure for $X$ and $Y$). Then we transform $Y[*, k_2, \ldots, k_d]$ by a 1D FFT for all tuples $(k_2, \ldots, k_d)$. This can be applied recursively to calculate a $d$-dimensional FFT by performing

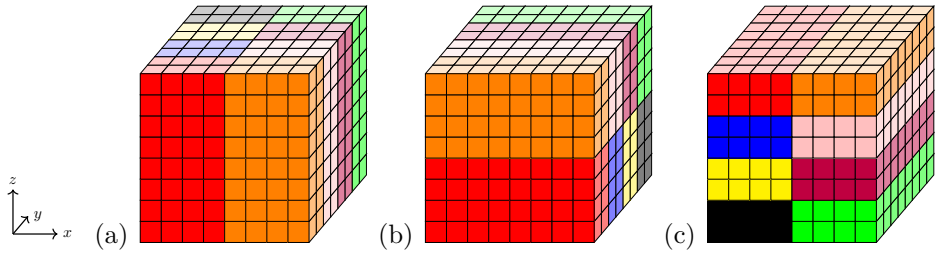



Figure 9.3: An $8 \times 8 \times 8$ array distributed over $2 \times 4$ processors in a pencil distribution along different dimensions. (a) Distribution along $x$ and $y$; (b) along $z$ and $y$; (c) along $x$ and $z$.

1D transforms along each dimension. This can be done in parallel using sequential 1D FFTs. See [78] for alternative parallel methods such as the binary exchange method and a comparison between these methods.

The approach that parallel FFTW [81] takes to parallelizing the multidimensional computation is as follows: it starts with a slab distribution, and performs a sequential FFT in all directions that are local. In 9.2(a), this is along the $y$-axis and the $z$-axis. Then it performs a communication step to move to a distribution such that the remaining direction becomes local, for instance the slab distribution of 9.2(b). Performing the final step in the slab distribution is not always possible. For example, for a parallel FFT on an array of size $8 \times 4 \times 2$ that starts with a slab distribution for 8 processors along the first dimension, the final step of FFTW uses a $4 \times 2$ pencil distribution. If needed, the final distribution will have an even higher dimension $r > 2$. Therefore, starting in a slab distribution along the first direction, with the largest size, FFTW can use at most $p_{\max} = \min(n_1, n_2 \cdots n_d) = \min(n_1, \frac{N}{n_1})$ processors. This implies that in the most favorable case, when $n_1 = \sqrt{N}$, FFTW can use $p_{\max} = \sqrt{N}$ processors, but in all other cases it must use fewer processors.

If we want to use more processors, we can choose to start in a pencil distribution instead of a slab distribution. This approach has first been proposed by Ding, Ferraro, and Gennery [66]. It has been implemented in 3D packages by Plimpton [193], see also [192], 2DECOMP&FFT by Li and Laizet [158] and P3DFFT by Pekurovsky [190]. PFFT [191] is also based on using a pencil distribution, but generalized to arbitrary $d \geq 3$.

Having a 2D processor distribution gives us $d-2$ directions that are locally available. This means we have to switch pencil distributions $\lceil \frac{d}{d-2} \rceil - 1 = \lceil \frac{2}{d-2} \rceil$ times to transform along all $d$ directions. For $d = 3$, this would mean switching twice, as illustrated by 9.3, and for $d \geq 4$, this would be only once. For $d = 3$, we can use $p_{\max} = \min(n_1 n_2, n_2 n_3, n_1 n_3) = n_2 n_3 = \frac{N}{n_1}$ processors, by our assumption on the decreasing sizes of the different dimensions. In the most favorable case, when all sizes are equal, $p_{\max} = N^{2/3}$; e.g., in 9.3, we could have used up to 64 processors.

PFFT generalizes earlier work by enabling the use of an $r$-dimensional data distribution with $1 \leq r < d$, not just $r = 1, 2$, at the expense of $\lceil \frac{d}{d-r} \rceil - 1 = \lceil \frac{r}{d-r} \rceil$

data redistributions. For $d \leq 3$, the slab and pencil distributions suffice; here, we discuss the remaining case $d \geq 4$. In general, distributing along $r$ dimensions makes the other $d - r$ dimensions locally available. Assume that the dimensions chosen for distribution have sizes $m_1, \ldots, m_r$ and the others have sizes $m_{r+1}, \ldots, m_d$; thus the $m_l$ are a permutation of the $n_l$. As a consequence, there are $m_1 \cdots m_r$ FFTs, which gives an upper bound on the number of processors we can use. We are left with $m_{r+1} \cdots m_d$ FFTs to perform, which reduces the upper bound to $\min(m_1 \cdots m_r, m_{r+1} \cdots m_d)$ processors. If we limit ourselves to requiring a single redistribution, this number equals $p_{\max}$, and in that case we must choose $r \leq d-r$, i.e., $r \leq d/2$. We can maximize $p_{\max}$ by choosing the dimensions to be distributed initially such that $m_1 \cdots m_r$ is as close as possible to $m_{r+1} \cdots m_d$. For even $d$, and in case all $n_l$ are the same, this means that $p_{\max} = N^{1/2}$ processors. For all other cases, $p_{\max} \leq N^{1/2}$ is an upper bound. For odd $d$, and in case all $n_l$ are the same, we can maximize $p_{\max}$ by taking $r = (d-1)/2$, giving $p_{\max} = N^{(d-1)/(2d)} < N^{1/2}$; e.g., for $d = 5$, $p_{\max} = N^{2/5}$. Thus, if all sizes are equal, having an odd number of dimensions is less favorable.

Both FFTW and PFFT finish their computation in a different distribution than the input distribution. If this is undesirable, an extra communication step is necessary. This may be avoided for instance when the FFT is followed by a local element-wise computation (such as happens in a convolution), an element-wise multiplication, and an inverse FFT. If this inverse FFT can start in the output distribution of the forward FFT and can finish in the original input distribution, then the extra communication step can be avoided.

Jung et al. [130] present a 3D FFT with a *volumetric* (3D) data distribution for use in an MD application. In one of their schemes, 1D all-to-all, they perform parallel 1D FFTs within every direction, with five all-to-all communication steps within one dimension. In an alternative scheme, 2D all-to-all, they require three all-to-all communication steps, each within one or two dimensions. Because of the requirements of the MD application of which the FFT is a part, the 3D distribution is by blocks.

Dalcin, Mortensen, and Keyes [59] provide new methods for accelerating the all-to-all communication step of multidimensional FFTs, where they make use of advanced features of MPI-2.

Popovici et al. [194] generalize the 1D FFT based on the cyclic distribution to a multidimensional FFT based on a cyclic distribution in all dimensions with the advantage that the input and output distribution are the same. They argue that to scale up to the large numbers of processors that are available in today's supercomputers, it it necessary to use parallelism both across 1D FFTs and within 1D FFTs. In each dimension $l$, their algorithm can then use $p_l \leq \sqrt{n_l}$ processors because of the cyclic distribution. This implies that if all $n_l$ are squares, the maximum number of processors used equals $p_{\max} = N^{1/2}$ . Their algorithm performs a communication step that moves all data, once for every dimension, so the algorithm has $d$ communication steps in total. In their implementation, they use FFTW for the local computations, MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism. They present results for 3D arrays of size up to $1024^3$.

### Contributions

This chapter provides the following contributions:

- We present a parallel multidimensional FFT algorithm based on the cyclic distribution that (i) has only a single all-to-all communication step; (ii) works for up to $p_{\max} = \sqrt{N}$ processors, where $N$ is the total number of array elements; (iii) starts and finishes in the same distribution.

- We present the program FFTU which is an implementation of this algorithm for an arbitrary dimension $d$ and present timing results for $d = 2, 3, 5$, demonstrating state-of-the-art performance by comparing to FFTW, PFFT, HeFFTe, and showing scalability for higher dimensions.

Starting and ending in the same distribution has the following advantages. First, the same program can be used for the forward and the inverse FFT, just with the weights conjugated and the outcome scaled by $1/N$, instead of reversing all the steps of the FFT. Furthermore, the inverse FFT can be performed directly after the FFT, possibly with an element-wise local operation interspersed, without the need for data reordering.

## 9.2 Generalized four-step framework

In this section, we first present the sequential four-step framework for the 1D FFT and derive from this a parallel cyclic-to-cyclic 1D FFT algorithm. We then generalize this algorithm to the multidimensional case, and prove that the resulting parallel multidimensional FFT algorithm does what it is supposed to do.

### One-dimensional algorithms

Suppose that $x$ is an array of length $n$, that $p|n$ and $p|\frac{n}{p}$ (or equivalently $p^2|n$), and that we want to calculate $y = F_n(x)$. We write $v(a : b : c)$ to mean the strided subarray of $v$ which starts at index $a$ and has stride $b$. The last variable $c$ is the length of the whole array $v$. If we write $v(a : b)$ we mean the subarray that starts at $a$ and ends at $b$ (inclusive).

To establish our notations and lay the basis for generalisation to the parallel and multidimensional case, we will first derive our basic sequential algorithm, the so-called *four-step framework*.

#### Sequential algorithm

As any $0 \leq j < n$ can be written uniquely in the form $kp + s$ with $0 \leq k < \frac{n}{p}$, $0 \leq s < p$, we have

$$y_a = \sum_{j=0}^{n-1} x_j \omega_n^{ja} = \sum_{s=0}^{p-1} \sum_{k=0}^{\frac{n}{p}-1} x_{kp+s} \omega_n^{(kp+s)a} = \tag{9.4}$$
$$\sum_{s=0}^{p-1} \omega_n^{sa} \sum_{k=0}^{\frac{n}{p}-1} x_{kp+s} (\omega_n^p)^{ka} = \sum_{s=0}^{p-1} \omega_n^{sa} \sum_{k=0}^{\frac{n}{p}-1} x_{kp+s} \omega_{\frac{n}{p}}^{ka}.$$

Write $x^{(s)}$ for $x(s : p : n)$ and $z^{(s)}$ for $F_{\frac{n}{p}}(x(s : p : n))$. We have the correspondence $x_k^{(s)} = x_{kp+s}$ and $x_j = x_{j \operatorname{div} p}^{(j \bmod p)}$ and likewise for $z$, $z^{(s)}$. Here, the div operator is defined by $j \operatorname{div} p = \lfloor j/p \rfloor$. By periodicity we have $\omega_{\frac{n}{p}}^{ka} = \omega_{\frac{n}{p}}^{k(a \bmod \frac{n}{p})}$. Armed with this knowledge we can state

$$\sum_{k=0}^{\frac{n}{p}-1} x_{kp+s} \omega_{\frac{n}{p}}^{ka} = \sum_{k=0}^{\frac{n}{p}-1} x_k^{(s)} \omega_{\frac{n}{p}}^{k(a \bmod \frac{n}{p})}. \tag{9.5}$$

But this is the definition of the $(a \bmod \frac{n}{p})$-th entry of the DFT of $x^{(s)}$, so we can write Eqn 9.4 as

$$y_a = \sum_{s=0}^{p-1} \omega_n^{sa} z_{a \bmod \frac{n}{p}}^{(s)}, \tag{9.6}$$

which looks suspiciously much like an FFT of length $p$. In order to massage this equation into that form, we write $a = t\frac{n}{p} + k$ for $0 \le k < \frac{n}{p}$, $0 \le t < p$,

$$y_{t\frac{n}{p}+k} = \sum_{s=0}^{p-1} \omega_n^{st\frac{n}{p}+sk} z_k^{(s)} = \sum_{s=0}^{p-1} \omega_p^{st} \left( z_k^{(s)} \omega_n^{sk} \right). \tag{9.7}$$

Fixing $k$, we see that $y(k : \frac{n}{p} : n) = F_p(w^{(k)})$, where $w_s^{(k)} = z_k^{(s)} \cdot \omega_n^{sk}$ for $0 \le s < p$.

We can summarize this as 16, which is known as the four-step framework and can be found with a different proof and formulation in [233]. Step 1 is called *twiddling* in [233] and elsewhere. We can easily avoid using extra arrays $z$ and $w$ by reusing $x, y$. The only step where it is not immediately obvious how to do this, is Step 3. Here, we note that $\{y(k : \frac{n}{p} : n) \mid 0 \le k < \frac{n}{p}\}$ splits $y$ into $\frac{n}{p}$ strided subarrays of length $p$ each, so we can use those to store the $\frac{n}{p}$ arrays $w^{(k)}$ of length $p$. Therefore, 16 can be executed mostly in place, where only in Step 2 special care has to be taken to limit the amount of extra memory needed to permute $z$ into $w$.

### Parallel algorithm

We use the Bulk Synchronous Parallel (BSP) model [232] as our approach to the parallelisation of the FFT. BSP algorithms comprise a sequence of *supersteps*, each performing a computation stage or a communication stage, where each superstep is terminated by a global synchronisation. Following the BSP approach in [27], we express our parallel algorithms in Single Program Multiple Data (SPMD) style,

**Algorithm 16** Sequential four-step framework

Input: $x$ : array of length $n$, number $p$ such that $p^2|n$.
Output: $y$ : array of length $n$, such that $y = F_n(x)$.

1: **for** $s := 0$ **to** $p-1$ **do** ▷ Step 0
2: $\quad x^{(s)} := x(s : p : n)$;
3: $\quad z^{(s)} := F_{\frac{n}{p}}(x^{(s)})$;
4: **for** $s := 0$ **to** $p-1$ **do** ▷ Step 1
5: $\quad$ **for** $k := 0$ **to** $\frac{n}{p}-1$ **do**
6: $\quad\quad z_k^{(s)} := \omega_n^{ks} z_k^{(s)}$;
7: **for** $s := 0$ **to** $p-1$ **do** ▷ Step 2
8: $\quad$ **for** $k := 0$ **to** $\frac{n}{p}-1$ **do**
9: $\quad\quad w_s^{(k)} := z_k^{(s)}$;
10: **for** $k := 0$ **to** $\frac{n}{p}-1$ **do** ▷ Step 3
11: $\quad y(k : \frac{n}{p} : n) := F_p(w^{(k)})$;

by giving the program text for processor $P(s)$ with $s \in [p]$. This means that although the executed program depends on $s$, we do not have to write different texts for different processors. We express the communication in our algorithm by the one-sided primitive 'Put', which completely defines the transferral of data by an action from the sender. The sender $P(s)$ states to which destination processor the data has to be sent, and to which memory address on that processor. The data can then be used in the next superstep.

We will analyse the time complexity of parallel algorithms within the BSP framework. The cost of a computation superstep in the BSP model depends on the maximum number of floating-point operations (flops) that a processor carries out, and we will measure these in real flops, as is usual. The cost of a communication superstep depends on the maximum number of data words each processor sends or receives, here complex numbers, and we will take $g$ (for *gap*) as the cost per data word. The cost of the synchronisation at the end of a superstep is taken as $l$ (for *latency*), but we will charge this cost only for communication supersteps. (The reason is that we only use Put operations in our program, not Get operations, and this means that in an actual implementation the synchronisation at the end of a computation superstep can be removed.) For more details on BSP, and an extensive treatment of the BSP 1D FFT with the cyclic distribution, see [27].

We will now derive a parallel version of the four-step framework. Step 0 and 1 of 16 are trivial to parallelize, and they determine the distribution: namely cyclic over $p$ processors. Here, processor $P(s)$ simply performs iteration $s$ of the outer loops. This yields Superstep 0 of the parallel 1D algorithm, 17. We would like to end in the same cyclic distribution in which we started. Let $y^{(s)} := y(s : p : n)$ be the part of the output vector $y$ stored by the cyclic distribution on $P(s)$. Component $c$ of this part is then denoted by $y^{(s)}[c] := y(s : p : n)[c]$.

We will first parallelize the final step of the sequential algorithm, Step 3,

aiming at a completely local operation in the cyclic distribution. Note that in the cyclic distribution over $p$ processors, global component $y_j$ is stored on processor $P(j \bmod p)$ in position $j \operatorname{div} p$. Consider component $c$ of $y(k : \frac{n}{p} : n)$, the output of the final step. Because $p|\frac{n}{p}$, we have that component $y(k : \frac{n}{p} : n)[c] = y_{c\frac{n}{p}+k}$ is stored on $P((c\frac{n}{p} + k) \bmod p) = P(k \bmod p)$ in local position $(c\frac{n}{p} + k) \operatorname{div} p = c\frac{n}{p^2} + k \operatorname{div} p$. So $y(k : \frac{n}{p} : n)$ is stored in its entirety on $P(k \bmod p)$. Taking $c = 0$, we see that $y(k : \frac{n}{p} : n)[0]$ is stored in local position $k \operatorname{div} p$. Furthermore, incrementing $c$ increases the local position by $\frac{n}{p^2}$. Therefore, $y(k : \frac{n}{p} : n) = y^{(k \bmod p)}(k \operatorname{div} p : \frac{n}{p^2} : \frac{n}{p})$, which is local on $P(s)$ for $k \bmod p = s$. As $k$ ranges from 0 to $\frac{n}{p} - 1$, we have that $t = k \operatorname{div} p$ ranges from 0 to $\frac{n}{p^2} - 1$. This determines the $k$-values for which the output of Step 3 is local, and we require the input also to be local. This yields Superstep 2, a local transformation of $y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p})$ by $F_p$ for $0 \leq t < \frac{n}{p^2}$.

Finally, we parallelize Step 2 of the four-step framework, which is a permutation of the data, and is usually called the *transposition* step. This step translates into a communication superstep that should transfer the components of the vector $w$ to their desired destination, so that the following superstep becomes local. Taking $c = s$ in the above, we obtain $y(k : \frac{n}{p} : n)[s] = y^{(k \bmod p)}(s\frac{n}{p^2} + k \operatorname{div} p)$. This means that we need to send the $x_k^{(s)}$ produced in Superstep 0 to $P(k \bmod p)$ in local position $s\frac{n}{p^2} + k \operatorname{div} p$. So $x^{(s)}(k : p : \frac{n}{p})$ ends up in $P(k)$ for $0 \leq k < p$. The first element corresponds to $y^{(k)}(s\frac{n}{p^2})$, and every time we increase $k$ by $p$, we increment $k \operatorname{div} p$, so $x^{(s)}(k : p : \frac{n}{p})$ corresponds to $y^{(k)}(s\frac{n}{p^2} : (s+1)\frac{n}{p^2} - 1)$. This is achieved by the Put-statement of Superstep 1. This completes our algorithm. If desired, we can make the algorithm completely in-place by taking $y = x$.

**Algorithm 17** Parallel four-step framework for $P(s)$

Input: $x$ : array of length $n$, distr$(x)$ = cyclic over $p$ processors such that $p^2|n$.
Output: $y$ : array of length $n$, distr$(y)$ = cyclic, such that $y = F_n(x)$.

1: $x^{(s)} := F_{\frac{n}{p}}(x^{(s)})$; ▷ Superstep 0
2: **for** $k := 0$ **to** $\frac{n}{p} - 1$ **do**
3: $\quad x_k^{(s)} := \omega_n^{ks} x_k^{(s)}$;
4: **for** $k := 0$ **to** $p - 1$ **do** ▷ Superstep 1
5: $\quad$ Put $x^{(s)}(k : p : \frac{n}{p})$ in $P(k)$ as $y^{(k)}(s\frac{n}{p^2} : (s+1)\frac{n}{p^2} - 1)$;
6: **for** $t := 0$ **to** $\frac{n}{p^2} - 1$ **do** ▷ Superstep 2
7: $\quad y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}) := F_p(y^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}))$;

## Parallel multidimensional algorithm

We will generalize 17 directly to a parallel multidimensional algorithm, 18, which we prove correct by showing that it computes the multidimensional DFT of Eqn

9.2. This equation can concisely be formulated in tensor notation, see [233], as

$$Y = (F_{n_1} \otimes \cdots \otimes F_{n_d})(X). \tag{9.8}$$

We use vector notation to abbreviate dimension-wise strides, subarrays, etc. So we define for instance $X(t : \frac{n}{p^2} : \frac{n}{p}) = X(t_1 : n_1/p_1^2 : n_1/p_1, \ldots, t_d : n_d/p_d^2 : n_d/p_d)$. The algorithm can be performed in-place using only an array $X$, but we have used different variables $X, Y, Z, W, V$ to facilitate the proof.

It is no coincidence that 18 combines instances of 17 dimension-wise. This happens because the multidimensional Fourier transform is the tensor product of 1D Fourier transforms and because we can combine BSP algorithms for linear functions into a BSP algorithm for the tensor product, as shown in [141]. The proof of this powerful observation involves category theory. Our correctness proof of 18, however, requires no such category theory, and is based on basic algebraic manipulations and concise notations.

**Algorithm 18** Parallel four-step framework for processor $P(s) = P(s_1, \ldots, s_d)$ in $d$ dimensions

Input: $X$ : multidimensional array of size $n_1 \times \cdots \times n_d$, distr$(X) = d$-dimensional cyclic over $p_1 \times \cdots \times p_d$ processors such that $p_l^2 | n_l$, for $l = 1, \ldots, d$.
Output: $V$ : multidimensional array of size $n_1 \times \cdots \times n_d$, distr$(V) = d$-dimensional cyclic, such that $V = (F_{n_1} \otimes \cdots \otimes F_{n_d})(X)$.

1: $Y^{(s)} := (F_{n_1/p_1} \otimes \cdots \otimes F_{n_d/p_d})(X^{(s)})$; ▷ Superstep 0
2: **for** $k \in [n_1/p_1] \times \cdots \times [n_d/p_d]$ **do**
3: $\quad Z^{(s)}[k] := (\prod_{l=1}^{d} \omega_{n_l}^{k_l s_l}) Y^{(s)}[k]$;
4: **for** $k \in [p_1] \times \cdots \times [p_d]$ **do** ▷ Superstep 1
5: $\quad$ Put $Z^{(s)}(k : p : \frac{n}{p})$ in $P(k)$ as $W^{(k)}[s\frac{n}{p^2} : (s+1)\frac{n}{p^2} - 1]$;
6: **for** $t \in [n_1/p_1^2] \times \cdots \times [n_d/p_d^2]$ **do** ▷ Superstep 2
7: $\quad V^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}) := (F_{p_1} \otimes \cdots \otimes F_{p_d}) \left( W^{(s)}(t : \frac{n}{p^2} : \frac{n}{p}) \right)$;

**Theorem 4.** *Algorithm 18 computes the multidimensional DFT expressed in Eqn 9.2.*

*Proof.* Applying the definition 9.2 of the multidimensional DFT for the local array of $P(s)$ gives

$$Y^{(s)}[k] = \sum_{j_1 \in [n_1/p_1]} \cdots \sum_{j_d \in [n_d/p_d]} X^{(s)}[j] \omega_{n_1/p_1}^{j_1 k_1} \cdots \omega_{n_d/p_d}^{j_d k_d}. \tag{9.9}$$

By using $\omega_{n_l/p_l}^{j_l k_l} = \omega_{n_l}^{j_l k_l p_l}$ and twiddling $Y^{(s)}$, we obtain

$$Z^{(s)}[k] = \sum_{j_1 \in [n_1/p_1]} \cdots \sum_{j_d \in [n_d/p_d]} X^{(s)}[j] \omega_{n_1}^{k_1(j_1 p_1 + s_1)} \cdots \omega_{n_d}^{k_d(j_d p_d + s_d)}, \tag{9.10}$$

which is the output of Superstep 0.

Now consider component $s'$ of the input array $W^{(s)}(t : \frac{n}{p^2} : \frac{n}{p})$ of Superstep 2. This component corresponds to $W^{(s)}[t + s'\frac{n}{p^2}] = W^{(s)}(s'\frac{n}{p^2} : (s'+1)\frac{n}{p^2} - 1)[t]$, which has been obtained by communication in Superstep 1 of component $Z^{(s')}(s : p : \frac{n}{p})[t] = Z^{(s')}[s + tp]$. This means that for $P(s)$ in the role of receiver of a block of data, $P(s')$ is the sender. Therefore,

$$W^{(s)}(t : \frac{n}{p^2} : \frac{n}{p})[s'] = Z^{(s')}[s + tp]$$
$$= \sum_{j_1 \in [n_1/p_1]} \cdots \sum_{j_d \in [n_d/p_d]} X^{(s')}[j]\, \omega_{n_1}^{(s_1+t_1p_1)(j_1p_1+s'_1)} \cdots \omega_{n_d}^{(s_d+t_dp_d)(j_dp_d+s'_d)}.$$

The final output $V^{(s)}(t : \frac{n}{p^2} : \frac{n}{p})$ of 18 is computed in Superstep 2. Consider component $k$ of the output array $V^{(s)}(t : \frac{n}{p^2} : \frac{n}{p})$, which can be written as $V^{(s)}[t + k\frac{n}{p^2}]$ and is computed by a local DFT,

$$V^{(s)}[t + k\frac{n}{p^2}] = \sum_{s'_1 \in [p_1]} \cdots \sum_{s'_d \in [p_d]} W^{(s)}(t : \frac{n}{p^2} : \frac{n}{p})[s']\, \omega_{p_1}^{s'_1k_1} \cdots \omega_{p_d}^{s'_dk_d}$$
$$= \sum_{s'_1 \in [p_1]} \cdots \sum_{s'_d \in [p_d]} \sum_{j_1 \in [n_1/p_1]} \cdots \sum_{j_d \in [n_d/p_d]} X^{(s')}[j]\, \omega_{n_1}^{(s_1+t_1p_1)(j_1p_1+s'_1)}$$
$$\cdots \omega_{n_d}^{(s_d+t_dp_d)(j_dp_d+s'_d)} \omega_{p_1}^{s'_1k_1} \cdots \omega_{p_d}^{s'_dk_d}. \quad (9.11)$$

To work towards the definition of the multidimensional DFT, we rewrite the powers of the roots of unity using $\omega_{p_l}^{s'_lk_l} = \omega_{n_l}^{s'_lk_ln_l/p_l} = \omega_{n_l}^{(j_lp_l+s'_l)k_ln_l/p_l}$. Note that the extra factor $\omega_{n_l}^{j_lp_lk_ln_l/p_l} = \left(\omega_{n_l}^{n_l}\right)^{j_lk_l} = 1$. This gives the expression

$$\sum_{s'_1 \in [p_1]} \cdots \sum_{s'_d \in [p_d]} \sum_{j_1 \in [n_1/p_1]} \cdots \sum_{j_d \in [n_d/p_d]} X^{(s')}[j] \omega_{n_1}^{(s_1+t_1p_1+k_1n_1/p_1)(j_1p_1+s'_1)} \cdots$$
$$\omega_{n_d}^{(s_d+t_dp_d+k_dn_d/p_d)(j_dp_d+s'_d)}. \quad (9.12)$$

Because we can write any number $j'_l \in [n_l]$ uniquely as $j'_l = s'_l + j_lp_l$ for $0 \le s_l < p_l$, $0 \le j_l < n_l/p_l$ and because of the local-global relations $X^{(s')}[j] = X[s' + jp]$ and $V^{(s)}[t + k\frac{n}{p^2}] = V[s + tp + k\frac{n}{p}]$, we obtain

$$V[s_1 + t_1p_1 + k_1\frac{n_1}{p_1}, \ldots, s_d + t_dp_d + k_d\frac{n_d}{p_d}] =$$
$$\sum_{j'_1 \in [n_1]} \cdots \sum_{j'_d \in [n_d]} X[j']\, \omega_{n_1}^{(s_1+t_1p_1+k_1n_1/p_1)j'_1} \cdots \omega_{n_d}^{(s_d+t_dp_d+k_dn_d/p_d)j'_d}. \quad (9.13)$$

This proves that the corresponding global array $V$ is precisely the image of the global array $X$ under the $d$-dimensional DFT. □

## Complexity analysis

We analyse the time complexity of our parallel multidimensional FFT algorithm within the BSP model [232]. Assume we have an array of size $n_1 \times \cdots \times n_d$. Let $N = n_1 \cdots n_d$ be the total number of array elements and $p = p_1 \cdots p_d$ be the total number of processors. (Note that we slightly abuse the notation, since we have used the variable $p$ in the previous section in a different meaning, namely as a tuple.) The sequential multidimensional FFT performs about $5N \log N$ flops for $N$ array elements, regardless of dimension or shape of the array. As the number of operations per array element is small, CPU–RAM bandwidth and cache architecture matter more than the precise number of flops, but we will disregard this in our theoretical analysis, and consider it part of the implementation concerns.

Let us first look at the computation supersteps of 18. The first part of Superstep 0 takes $5\frac{N}{p} \log \frac{N}{p}$ flops, and Superstep 2 takes $\frac{N}{p^2} \cdot 5p \log p$. With proper implementation, the twiddling takes about two complex multiplications per element, as we will see in 19 in the next section. So this adds $12\frac{N}{p}$ real flops, giving a total number of flops of

$$T_{\text{comp,FFT}} = 5\frac{N}{p} \log \frac{N}{p} + 5\frac{N}{p} \log p + 12\frac{N}{p} = 5\frac{N}{p} \log N + 12\frac{N}{p}. \tag{9.14}$$

Except for the additional $12\frac{N}{p}$ flops from the twiddling, the original sequential workload is perfectly divided into $p$ parts.

Looking at the communication superstep, Superstep 1, we note that each array element is communicated at most once, with each processor sending and receiving at most $\frac{N}{p}$ elements, at a cost of $\frac{N}{p}g$. This is a balanced all-to-all communication step. Since we have only one communication superstep for which we charge the synchronisation, the total synchronisation cost of the algorithm is $l$. Therefore, the total BSP cost of our parallel multidimensional algorithm is

$$T_{\text{FFT}} = 5\frac{N}{p} \log N + 12\frac{N}{p} + \frac{N}{p}g + l. \tag{9.15}$$

Our algorithm has a scalability limit for the maximum number of processors $p_{\max}$ that can be used, because in dimension $l$ we can use up to $\sqrt{n_l}$ processors; this is in the ideal case that $p_l^2 | n_l$. If all $n_l$ are squares, this limit equals

$$p_{\max} = (n_1 \cdots n_d)^{1/2} = \sqrt{N}. \tag{9.16}$$

If not all $n_l$ are squares, the maximum is a bit lower. In case the $n_l$ and $p_l$ are powers of two, $p_{\max}$ is then lowered by a factor of 2 for every $n_l$ that is not a power of four. For a 3D array of size $1024^3$, our algorithm can use up to $32^3 = 32{,}768$ processors. For 3D arrays of size $256^3$ and $512^3$, our algorithm can use up to $16^3 = 4096$ processors. For a 2D array of size $2^{24} \times 64$, with the same number of elements as the array of size $1024^3$ but with a very high aspect ratio, we can still use the same number of processors, $p_{\max} = 32{,}768$.

It is possible to scale beyond $p_{\max} = \sqrt{N}$, but in that case more than one communication superstep is needed and a generalisation of the cyclic distribution

must be used, called the group-cyclic distribution [120], see also [27] for an explanation and an implementation for the 1D FFT. The *group-cyclic distribution* with cycle $c$ splits an array of size $n$ into $\frac{p}{c}$ blocks of size $\frac{cn}{p}$. It assigns each block to a group of $c$ processors and it uses the cyclic distribution within each block. In this distribution, array element $x_j$ is assigned to processor $P((j \operatorname{div} \frac{cn}{p})c + j \bmod c)$. This is a different generalisation of the cyclic distribution than the well-known block-cyclic distribution, which is used in certain parallel numerical linear algebra packages, for instance in ScaLaPack [29].

## 9.3 Implementation

In this section, we present the implementation of our multidimensional FFT algorithm, which we humbly call FFTU, the fastest FFT in Utrecht. The implementation can be found at `https://gitlab.com/Thomas637/FFT` and it contains both an MPI implementation and a BSPlib implementation. FFTU has been released under the GNU GPL license. We discuss only the MPI implementation since the BSPlib version is very similar. The only difference is that in the BSPlib version we unpack the data packets manually instead of letting the communication library do this.

We use FFTW for the local computations and MPI for communication. We move the data into a buffer so that the data to be sent to a single processor is stored contiguously; this is called *packing*. We have implemented a method where we send the packets using MPI_alltoall, and then locally rearrange the data, moving them into the correct positions (called *unpacking*), and also a method using MPI_alltoallv, where we specify how the packet is to be stored on the remote processor using derived data types. This allows (but does not require) the MPI implementation to perform the communication without local data movement.

We combine the packing with the twiddling to minimize the consumption of CPU–RAM bandwidth. This gives 19, which can be applied to the local array $X^{(s)}[k]$. Note that we perform only two complex multiplications per data element in the innermost loop of the algorithm, so that the time complexity is about $12N/p$ real flops.

The twiddle weights $\omega_{n_l}^{k_l s_l}$ used in the FFT algorithm can be precomputed and stored in a weight table, which requires a local memory of

$$M_{\text{twiddle}} = \sum_{l=1}^{d} \frac{n_l}{p_l}, \tag{9.17}$$

which is much less than the $\prod_{l=1}^{d} \frac{n_l}{p_l} = \frac{N}{p}$ memory needed to store the local array $X^{(s)}$.

For the sequential multidimensional FFTs that we use in 18, numerous optimized libraries are available, such as FFTW [81] and SPIRAL [200]. We have chosen for FFTW as it has the most flexible interface, which we exploit for the interleaved strided arrays, but our algorithm can also easily be adapted to use SPIRAL or other libraries. The row-major format of the input and output means that

**Algorithm 19** Packing and twiddling

Input: a $d$-dimensional array $X$ of size $n_1/p_1 \times \cdots \times n_d/p_d$ in row-major format, $s = (s_1, \ldots, s_d) \in [p]$.
Output: $d$-dimensional arrays $\text{packet}_k$ containing the twiddled $X(k : p : \frac{n}{p})$ in row-major format.

1: **for** $t_1 := 0$ **to** $n_1/p_1 - 1$ **do**
2: $\quad \text{factor}_1 := \omega_{n_1}^{t_1 s_1}$;
3: $\quad$ **for** $t_2 := 0$ **to** $n_2/p_2 - 1$ **do**
4: $\quad\quad \text{factor}_2 := \text{factor}_1 \cdot \omega_{n_2}^{t_2 s_2}$
5: $\quad\quad \vdots$
6: $\quad\quad$ **for** $t_d := 0$ **to** $n_d/p_d - 1$ **do**
7: $\quad\quad\quad \text{factor}_d := \text{factor}_{d-1} \cdot \omega_{n_d}^{t_d s_d}$;
8: $\quad\quad\quad \text{packet}_{t \bmod p}[t \operatorname{div} p] := X[t] \cdot \text{factor}_d$;

the last dimension is consecutive in memory, like in the programming language C.

## 9.4 Numerical experiments

### Setup

We performed strong scaling experiments for three different programs: our program FFTU, FFTW, PFFT, HeFFTe. All are compiled with Intel 2021.2.0, which includes Intel MPI, and flags -O3 -march=native unless otherwise specified. The FFTW version is 3.3.9. Finally we use PFFT commit e4cfcf9 on `https://github.com/mpip/pfft` (no version number available). For $p = 1, 2$, the derived data type version of FFTU with Intel MPI does not terminate. For this reason, we used the manual unpacking method for $p = 1, 2$. We use HeFFTe version 2.2 with Intel's Math Kernel Library version 2022.1.0.

FFTW can generate faster FFT functions at the cost of some setup time. This is controlled by a flag, offering choices FFTW_ESTIMATE, FFTW_MEASURE, FFTW_PATIENT. We tested these locally on an array of size $256 \times 256 \times 256$ and the execution (setup) time took 2.331 (0.03), 0.176 (2.735), 0.170 (239) seconds respectively.
As FFTW_PATIENT only pays off after about $40,000$ executions, we chose to use FFTW_MEASURE. In HeFFTe the use of FFTW_ESTIMATE is hard coded, and replacing this by FFTW_MEASURE leads to runtime errors. For this reason we use Intel's Math Kernel Library, which is also the sequential time reported for HeFFTe.

We ran our timing experiments on arrays with a total number of $N = 2^{30}$ elements, in shapes $1024^3$, $64^5$ for all three programs, and in shape $16{,}777{,}216 \times 64$ for FFTU and FFTW. We obtained timings on the thin node partition of the supercomputer Snellius at SURFsara in Amsterdam. A thin node consists of two 64-core AMD Rome 7H12 processors in a dual socket configuration, running at 2.6

GHz. There is 2 GiB of RAM available per core. The interconnect is Infiniband HDR100 (100 Gbps) with a fat tree topology. The job scheduler used is SLURM, and we do not share nodes. The operating system is CentOS7. We have used a single switch in our experiments and bound the MPI processes to cores with −−cpu_bind=cores.

Ideally, we would write down the time at the start of the FFT, at the end, and subtract the two values to obtain a timing result. The problem with this method is that the synchronizing function MPI_Barrier of the MPI library only guarantees that no processor leaves the barrier before all processors have entered. So there is no guarantee that every processor starts the FFT function at the same time. That is why we apply the FFT 100 times, so that the small difference in leaving the barrier becomes negligible. For HeFFTe we use the benchmark program speed3d_c2c provided by them.

For FFTW and PFFT, we measure both the time it takes when we demand to end in the same distribution that we started in (default for FFTW and with flag PFFT_TRANSPOSED_NONE for PFFT), and when we allow to end in a different distribution (flags FFTW_TRANSPOSED_OUT and PFFT_TRANSPOSED_OUT, respectively). HeFFTe does not provide an option to end in the same distribution.

## Results

Table 9.1 presents the timing results for an FFT of an array of size $1024^3$ for different numbers of processors $p$. This represents the 3D problem, which is likely the most common application of a multidimensional FFT, used in modeling our 3D physical space. If we require the input and output to be in the same distribution, we see that FFTU performs better than PFFT in all cases, and better than FFTW for $p \geq 128$. For smaller $p$, FFTW outperforms both PFFT and FFTU, which invoke FFTW but entail additional overhead. This is particularly clear from the large parallel overhead for $p = 1$ when compared to the sequential case, which represents a slowdown of a factor 2.3 for FFTU and 2.9 for PFFT. Once we exceed the number of cores in a socket, i.e. for $p > 64$, communication becomes more costly and the advantage of fewer communication supersteps becomes clear. The highest speedup achieved for FFTU compared to sequential FFTW is 149× on 4096 processors, and for PFFT it is 98.5×. FFTW can use only 1024 processors, with a speedup of 32.1×. HeFFTe achieves a speedup of 84× relative to Intel's Math Kernel Library. Counting $5N \log N$ flops to transform an array of $N$ elements, FFTU reaches a top computing rate of 0.94 Tflop/s for this problem.

If we allow the input and output distribution to be different, both PFFT and FFTW can save the final communication superstep. For this 3D problem, all three programs perform a single communication superstep for $p \leq 1024$, and FFTU does this for all given $p$, because it could go up to $p = 32{,}768$ with a single communication superstep. For $p > 1024$, PFFT needs $\lceil \frac{3}{3-2} \rceil - 1 = 2$ communication supersteps, because it then uses a 2D decomposition, putting as many processors along the first dimension as possible. For FFTW, the maximum number of processors that can be used is 1024.

PFFT with output allowed in a different distribution is about equally fast as FFTU. It is surprising that this also holds for $p = 2048, 4096$, given the extra superstep PFFT needs. When inspecting the instance $p = 4096$ for the case with the same distribution imposed, we noticed that the final communication superstep performing the redistribution takes as long as the entire calculation, where we would expect it to be less than half. This means that PFFT performs its internal two communication supersteps very efficiently. It does this by using the global transposes of FFTW instead of directly calling MPI, and optimizing these by precomputing several FFTW plans, see [191, Section 3.3]. Furthermore, we observe a superlinear speedup of PFFT for $p = 4096$ compared with $p = 2048$.

In 9.1, we see that FFTW with output allowed in a different distribution is faster than FFTU. This can be explained by FFTW doing a better job at communicating in its communication superstep, perhaps by better exploiting the shared memory available. We performed an additional experiment to see whether we could improve the performance of FFTU. Using OpenMPI version 4.1.1 instead of Intel MPI, the time needed decreased from 0.664 s to 0.515 s for $p = 512$, actually even beating FFTW. So there is room for improvement in the MPI implementation and it might be possible to attain the same performance as FFTW.

Although we cannot compare the absolute running times of HeFFTe to FFTU due to the different sequential library, we can compare the way it scales. We see good scaling even when going to 2048, where an extra communication step is needed. Like PFFT, HeFFTe can hide this communication well. However, we do not see superlinear speedup when going to 4096 processors for HeFFTe, instead showing similar speedup to FFTU going from 2048 to 4096 processors.

Table 9.2 shows the results for an array of size $64^5$. Here, FFTW can only use up to 64 processors, so there are no timings for FFTW beyond this number. Comparing the 5D timings of this table with the 3D timings of 9.1, we see that higher-dimensional FFTs run faster for the same total number of array elements. The highest speedup achieved by FFTU is 176× and that of PFFT is 225×.

In 9.2, we see that FFTU performs about as well as PFFT when we do not require the output to be in the same distribution as the input. This makes sense, as a 2D processor distribution suffices, meaning we need $\lceil \frac{5}{5-2} \rceil - 1 = 1$ communication superstep. For $p = 128, 512, 1024, 2048$ FFTU is a bit faster, but for $p = 256, 4096$ this is the other way round. These differences can be explained by the implementation of the communication step. We simply describe to MPI what we want by use of MPI_Alltoallv and derived data types, and let the black box MPI library handle the exact algorithm for communication. FFTW and PFFT experiment with several algorithms during the planning phase and then choose the fastest option. Apparently, sometimes MPI chooses the superior algorithm, and sometimes FFTW/PFFT does. We again observe superlinear speedup for PFFT when going from 2048 to 4096 processors. The same happens when doubling the array size to $128 \times 64^4$, so we presume that this phenomenon is not due to cache size.

Table 9.3 shows timing results for an array of size $16{,}777{,}216 \times 64$. This problem with a high aspect ratio failed for PFFT, because of an integer division-by-zero error. We did not investigate this case further and just present timings for FFTU

Table 9.1: Time (in s) of a multidimensional FFT for a $1024^3$ array using three programs: FFTU, PFFT, and FFTW. The program FFTU yields the same data distribution for the output as for the input. For PFFT and FFTW, this can be imposed ("same") or not ("different"). For comparison, the time for running the sequential FFTW is also given.

| $p$ | FFTU | PFFT | | FFTW | | HeFFTe |
|---|---|---|---|---|---|---|
| | same | same | different | same | different | different |
| seq | | | | 17.541 | | 32.834 |
| 1 | 40.065 | 51.334 | 21.646 | 23.025 | 19.615 | - |
| 2 | 18.058 | 27.562 | 12.359 | 13.650 | 12.519 | 18.385 |
| 4 | 8.074 | 13.179 | 6.432 | 6.962 | 6.236 | 15.354 |
| 8 | 3.999 | 9.102 | 4.290 | 4.024 | 3.260 | 8.167 |
| 16 | 2.349 | 5.552 | 2.510 | 2.388 | 1.803 | 5.409 |
| 32 | 1.789 | 3.190 | 1.417 | 1.545 | 1.145 | 3.589 |
| 64 | 1.802 | 3.133 | 1.411 | 1.670 | 1.378 | 2.814 |
| 128 | 1.366 | 3.330 | 1.461 | 1.996 | 1.475 | 2.782 |
| 256 | 0.980 | 1.972 | 0.918 | 1.208 | 0.797 | 1.905 |
| 512 | 0.664 | 1.409 | 0.677 | 0.991 | 0.577 | 1.236 |
| 1024 | 0.317 | 0.644 | 0.327 | 0.546 | 0.310 | 0.618 |
| 2048 | 0.163 | 0.417 | 0.223 | | | 0.393 |
| 4096 | 0.118 | 0.178 | 0.088 | | | 0.277 |

and FFTW. Note that both PFFT and FFTU can use at most 64 processors for this problem, since they need one dimension to be local.

FFTU is slower here than in the 3D and 5D problems with the same $N$. Our explanation is as follows. The total size of the twiddle tables is $\sum_{l=1}^{d} \frac{n_l}{p_l}$, see Eqn 9.17, so in the case where one of the $n_l$ is relatively close to $N$ and the others are small, this is a relatively large table, which cannot be stored in cache and hence leads to a performance penalty. The twiddle tables are computed during the algorithm and then repeatedly used; we found no need for precomputing the twiddle weights, as is done for the other weights by FFTW.

## 9.5 Conclusions

We have developed an efficient and scalable multidimensional FFT algorithm that has a single all-to-all communication superstep, that starts and ends in the same distribution, and that can use up to $\sqrt{N}$ processors (or slightly less if the array size in some dimensions is not a square number). We have implemented this algorithm in a program FFTU and tested it on a distributed-memory parallel architecture, comparing it to two state-of-the-art programs, PFFT and FFTW. Our program FFTU uses FFTW for its local computations and it calls MPI for its communication. We can view our package FFTU as an extension of FFTW, in

Table 9.2: Time (in s) of a multidimensional FFT for a $64^5$ array using three programs: FFTU, PFFT, and FFTW. The program FFTU yields the same data distribution for the output as for the input. For PFFT and FFTW, this can be imposed ("same") or not ("different"). For comparison, the time for running the sequential FFTW is also given.

| $p$ | FFTU | PFFT | | FFTW | |
|---|---|---|---|---|---|
| | same | same | different | same | different |
| seq | | | | 17.381 | |
| 1 | 36.334 | 23.981 | 16.134 | 18.803 | 19.451 |
| 2 | 17.843 | 14.548 | 9.844 | 12.690 | 11.738 |
| 4 | 7.771 | 7.630 | 5.053 | 6.826 | 6.130 |
| 8 | 4.111 | 4.226 | 2.746 | 3.538 | 3.148 |
| 16 | 2.372 | 2.669 | 1.614 | 2.119 | 1.862 |
| 32 | 1.653 | 2.165 | 1.125 | 1.593 | 1.301 |
| 64 | 1.634 | 2.259 | 1.222 | 1.390 | 0.997 |
| 128 | 1.315 | 2.735 | 1.551 | | |
| 256 | 0.965 | 1.650 | 0.956 | | |
| 512 | 0.609 | 1.256 | 0.667 | | |
| 1024 | 0.304 | 0.644 | 0.357 | | |
| 2048 | 0.167 | 0.358 | 0.190 | | |
| 4096 | 0.099 | 0.159 | 0.077 | | |

the same way as PFFT can be considered an extension.

Overall, FFTU performs on par with PFFT and FFTW if the output distribution need not be the same as the input distribution and FFTU outperforms PFFT and FFTW if these distributions must be the same. There are some fluctuations in the behavior of the three programs, which we cannot completely explain, because of shared-memory effects and highly optimized implementations of PFFT and FFTW incorporating multiple algorithms for global communication.

The most important application of our algorithm will most likely be the 3D case, because of the single all-to-all communication superstep where PFFT and FFTW need two or even three in case the output distribution needs to be the same as the input distribution. Another important application would be the case where the input array is very rectangular, with one dimension much larger in size than the others. Here, the advantage is better scalability, because we can still use $\sqrt{N}$ processors, where the other methods are limited by the size of the smallest dimensions.

## 9.6 Future work

Our package FFTU performs multidimensional complex-to-complex fast Fourier transforms in the cyclic distribution. For future work, this could be extended

Table 9.3: Time (in s) of a multidimensional FFT for a $16{,}777{,}216 \times 64$ array using two programs: FFTU and FFTW. The program FFTU yields the same data distribution for the output as for the input. For FFTW, this can be imposed ("same") or not ("different"). For comparison, the time for running the sequential FFTW is also given.

| $p$ | FFTU | FFTW | |
|---|---|---|---|
| | same | same | different |
| seq | | 24.182 | |
| 1 | 43.146 | 26.984 | 31.440 |
| 2 | 21.950 | 16.661 | 17.382 |
| 4 | 9.613 | 8.649 | 8.563 |
| 8 | 5.150 | 4.577 | 4.609 |
| 16 | 3.045 | 2.695 | 2.699 |
| 32 | 2.347 | 2.023 | 1.959 |
| 64 | 2.218 | 1.646 | 1.442 |
| 128 | 1.615 | | |
| 256 | 1.264 | | |
| 512 | 0.841 | | |
| 1024 | 0.331 | | |
| 2048 | 0.230 | | |
| 4096 | 0.204 | | |

to related transforms such as the real-to-complex Fourier transform (RFFT), the discrete sine transform (DST), and the discrete cosine transform (DCT). We can do this by combining existing 1D algorithms dimension-wise [141]. To exploit the symmetry of these transforms, the *zig-zag cyclic distribution* [120, 27] could be used instead of the cyclic distribution.

All our parallelism is explicit and based on the distributed-memory approach using MPI, and we do not yet exploit the possible improvements shared-memory parallelism has to offer. With 128 cores on a processor node, this may yield significant improvements in our communication superstep. Using OpenMP, or using some of the communication schemes of FFTW for this purpose could be beneficial. Another research direction would be to exploit better methods for performing the single communication superstep in our algorithm, either by letting the transposition algorithms of FFTW handle this superstep, or by using improved redistribution methods [59] instead of our vanilla call to MPI.

Some applications perform element-wise multiplications in both domains of the Fourier transform. In the solution of the time-dependent Schrödinger equation on a multidimensional grid, a wave function is propagated in time by multiplying it point-wise by a potential function in the time domain, and by a momentum operator in the frequency domain. This means that we only need one all-to-all communication superstep per (forward or inverse) transform. No further communication is required in the propagation.

In other applications, however, such as classical molecular dynamics, the data may have to be stored in a block distribution instead of a cyclic distribution, to accommodate parts of the application program outside the FFT. This may require additional data redistribution; further investigation is needed to see how this can be avoided or mitigated.

# Chapter 10

# Shape-Guided Matrix Multiplication

## 10.1 Introduction

Matrices are representations of linear functions between vector spaces. Much of scientific computing concerns itself with linear algebra. If $A$ is a matrix that represents $f$ and $B$ a matrix that represents $B$, then the matrix product $AB$ represents $f \circ g$. This makes matrix multiplication an important building block in linear algebra. The performance of its straightforward implementation is limited by moving data from main memory to caches, and caches to registers, rather than computation. This bandwidth bottleneck can be removed by doing as many computations as possible on a block of memory that fits in a fast(er) level of the memory hierarchy, called a blocked algorithm.

The algorithmic idea behind a blocked matrix multiplication can be summarized in one word: recursion. This simple idea is the basis of efficient matrix multiplication on a range of architectures, from CPUs [91] to distributed systems [84]. As simple the idea, so complicated the implementations. There are two reasons for this unsatisfying contrast. First, languages such as C and Fortran cannot express the algorithmic idea. Second, the blocking that optimizes performance requires deep understanding of the target hardware.

This chapter shows we can eliminate this contrast by raising the abstraction level of the implementation language. The contribution of this chapter is an elegant expression of the recursive idea by using arrays of arbitrary rank, which specifies the blocking in the shape of the argument, rather than the function implementation itself (Listing 10.4). Though this does not solve the second problem of understanding the hardware, it does isolate this problem to choosing the shape of the input arguments, which we demonstrate in Section 10.4.

## 10.2 Matrix Multiplication

Suppose $A$ is an $m \times k$ matrix and $B$ is an $k \times n$ matrix. Then the canonical definition of their product $C$ is given by the following formula for element at index $(i, j)$:

$$C_{i,j} = \sum_{p<k} A_{i,p} \cdot B_{p,j}.$$

The product $C$ represents the composition of the linear functions that $A$ and $B$ represent.

### Bandwidth Bottleneck

We can understand the performance bottleneck of a straightforward implementation by looking at the ratio between computations and data movement. A computation $C_{i,j} = C_{i,j} + A_{i,p} \cdot B_{p,j}$ can be executed in one instruction (called a fused multiply-add, or fma), and we need $mkn$ of these in total. On our target machine, various levels of parallelism allow us to do 256 fma operations per clock cycle. However, we can load less than four elements of $B$ per clock cycle. To compute one row of $C$, we need to access all of $B$, so this results in $mkn$ total data movements. The 1: 1 ratio of computation and data movement makes data movement the bottleneck when computing $C$ row-by-row, assuming $B$ does not fit in cache. Computing $C$ column-by-column gives the same problem for $A$.

### Blocked Matrix Multiplication

We can split $A$ and $B$ into blocks, and define a blocked version of the same algorithm. Given factorisations $m = \hat{m}b_m$, $k = \hat{k}b_k$ and $n = \hat{n}b_n$, we can subdivide $A$ in blocks of size $b_m \times b_n$, $B$ into blocks of size $b_n \times b_k$ and $C$ into blocks of size $b_m \times b_n$. In this case, matrix multiplication can be expressed recursively as:

$$\begin{pmatrix} A_{0,0} & \cdots & A_{0,\hat{k}-1} \\ \vdots & \ddots & \vdots \\ A_{\hat{m}-1,0} & \cdots & A_{\hat{m}-1,\hat{k}-1} \end{pmatrix} \begin{pmatrix} B_{0,0} & \cdots & B_{0,\hat{k}} \\ \vdots & \ddots & \vdots \\ B_{\hat{k}-1,0} & \cdots & B_{\hat{k}-1,\hat{n}-1} \end{pmatrix} =$$

$$\begin{pmatrix} \sum_{p<\hat{k}} A_{0,p}B_{p,0} & \cdots & \sum_{p<\hat{k}} A_{0,p}B_{p,\hat{n}} \\ \vdots & \ddots & \vdots \\ \sum_{p<\hat{k}} A_{\hat{m}-1,p}B_{p,0} & \cdots & \sum_{p<\hat{k}} A_{\hat{m}-1,p}B_{p,\hat{n}-1} \end{pmatrix}$$

The interesting observation here is that we can iterate this process by applying blocking within the inner matrix multiplications. We can do this up to the moment we get to the matrices of size $1 \times 1$, in which we can stop and use a regular element product to compute the result. Putting these observations together we obtain a fully recursive algorithm where the base case is a regular product, and the recursive step is the blocked decomposition described above.

The important difference with the non-recursive specification is that the update $C_{i,j} = C_{i,j} + A_{i,p}B_{p,j}$ multiplies matrices instead of scalars. The matrix

```
double[m, n] matmul(double[m, k] A, double[k, n] B)
{
    return {[i, j] -> sum({[p] -> A[i, p] * B[p, j]})};
}
```

Listing 10.1: Canonical Matrix Multiplication in SaC

```
double[m, n] sum_2d(double[k, m, n] Cs)
{
    return {[i, j] -> sum({[p] -> Cs[p, i, j]})};
}

double[m1, n1, m2, n2] matmul(double[m1, k1, m2, k2] A,
                              double[k1, n1, k2, n2] B)
{
    return {[i, j] ->
              sum_2d({[p] -> matmul(A[i, p], B[p, j])})};
}
```

Listing 10.2: One level of blocking in SaC

multiplication has runtime complexity $b_m b_k b_n$, and a block of $B$ has size $b_k \times b_n$, so the ratio between computation and data movement is now $b_m : 1$, rather than $1 : 1$. This eliminates the bandwidth bottleneck if we can choose $b_m$ large enough while keeping $B$ in fast memory.

## 10.3 Shape Recursion in SaC

We now implement the blocked algorithm in SaC, assuming some familiarity, otherwise consider [210, 221] as an introduction to the language.

### Rank Polymorphism

The mathematical definition of matrix multiplication can be straightforwardly translated into the SaC program of Listing 10.1. This function takes two-dimensional arrays, and the size is described in the types double[m, k], double[k, n].

We can introduce a level of blocking by adding a matmul function on four-dimensional arrays, as in Listing 10.2. This is not very satisfying however, as we would need to keep adding functions for every level of blocking, despite the versions not being fundamentally different.

To implement any level of blocking elegantly, we need a feature called *rank-polymorphism*, or the ability to write one function that can take an array of any dimension (also called rank). We can describe this with double[d:shp]. In this

```
double[d:shp] rsum(double[k, d:shp] Cs)
{
    return {iv -> sum({[p] -> Cs[[p]++iv] | [p] < [k]})
                | iv < shp};
}
```

Listing 10.3: Rank-Polymorphic summation

```
double matmul(double A, double B)
{
  return A * B;
}

double[m, n, d:shpc]
matmul(double[m, k, d:shpa] A, double[k, n, d:shpb] B)
{
  return {[i, j] ->
             rsum({[p] -> matmul(a[i, p], b[p, j])})};
}
```

Listing 10.4: Blocked Matrix Multiplication in SaC

case, we would have d equal to two, and shp equal to [M, K] or [K, N]. With this, we can generalize sum_2d to a ranked summation rsum (Listing 10.3). Here index [i, j] is replaced by an index vector iv, and the concatenation [p, i, j] of [p] and [i, j] is made explicit by the ++ function.

## Recursive Matrix Multiplication

We now have all ingredients to specify a recursive matrix multiplication on blocked arrays in SaC, Listing 10.4. The function matmul is overloaded, and the recursion will stop when $A$ and $B$ have rank 0 (are scalars).

The rank-polymorphism of SaC allows us to describe the level of blocking in the shape of the arguments, rather than in the implementation of the matrix multiplication. The compiler can then transform this into an equivalent loop nest. For reference, a three-level blocking in C looks like Listing 10.5.

## Blocking

The arguments of the blocked matrix multiplication are assumed to already be in a higher-dimensional blocked form. This raises the question of how we transform between arrays of dimension two and higher dimensions. If we have decompositions

$$m = m_1 \cdot \cdots \cdot m_d, \quad k = k_1 \cdot \cdots \cdot k_d,$$

```
for (int i1 = 0; i1 < m; i1 += bm1)
  for (int j1 = 0; j1 < n; j1 += bn1)
    for (int p1 = 0; p1 < k; p1 += bk1)
      for (int i2 = i1; i2 < i1 + bm1; i2 += bm2)
        for (int j2 = j1; j2 < j1 + bn1; j2 += bn2)
          for (int p2 = p1; p2 < p1 + bk1; p2 += bk2)
            for (int i3 = i2; i3 < i2 + bm2; i3++)
              for (int j3 = j2; j3 < j2 + bn2; j3++)
                for (int p3 = p2; p3 < p2 + bk2; p3++)
                  C[i3 * n + j3] +=
                  A[i3 * k + p3] * B[p3 * n + j3];
```

Listing 10.5: Blocked Matrix Multiplication in C

we can create a $d$-dimensional array $A$ of shape [m1, ..., md, k1, ..., kd] by calling reshape([m1, ..., md, k1, ..., kd], A), but this computes the wrong result. We can repair this by calling the function block of Listing 10.6 after the reshape (and unblock to get a 2-dimensional array back). These functions look strange, especially the reshapes to [2, d / 2]. In the original paper [219] we have shown mechanically that this is correct, so in this subsection we will focus on the mathematical intuition.

```
int[d] new_shape(int[d] iv) {
    return reshape([d], transpose(reshape([2, d/2], iv)));
}

int[d] old_shape(int[d] iv) {
    return reshape([d], transpose(reshape([d/2, 2], iv)));
}

double[d:shpo] block(double[d:shpi] A) {
    return {iv -> A[old_shape(iv)] | iv < new_shape(shpi)};
}

double[d:shpo] unblock(double[d:shpi] A) {
    return {iv -> A[new_shape(iv)] | iv < old_shape(shpi)};
}
```

Listing 10.6: Blocking in SaC

The source of confusion is that we have two different types of mathematical objects that are represented by arrays: linear functions and vectors. A vector space over $\mathbb{R}$ consists of a basis $(e_i \mid i \in I)$ and all formal linear combinations $\{\sum_{i\in I} X_i e_i \mid i \in I\}$. If $I = \{0, \cdots, n-1\}$, this defines an array X of shape n, which is how vectors are usually represented. But we can also take $I$ equal to $\{0, \cdots, n_1 - 1\} \times \cdots \times \{0, \cdots, n_d - 1\}$, which defines a $d$-dimensional array. The canonical way of transforming an index space from $d$-dimensional to 1-dimensional

in C—and therefore also SaC—is row-major order (Fig. 10.1). This is done by the reshape function.

$$\begin{pmatrix} e_0 = f_{0,0} & e_1 = f_{0,1} & e_2 = f_{0,2} \\ e_3 = f_{1,0} & e_4 = f_{1,1} & e_5 = f_{1,2} \end{pmatrix}$$

Figure 10.1: Row-major correspondence between basis $e$ with index set $\{0, \cdots, 5\}$, and basis $f$ with index set $\{0, 1\} \times \{0, 1, 2\}$

An $m \times n$ matrix however, does not represent a vector, but a linear map. Let $V$ be a vector space with basis $(e_1, \cdots, e_m)$ and $W$ a vector space with basis $(f_1, \cdots, f_n)$. Then matrix $A$ represents linear map $a(e_i) = \sum_{j=1}^{n} A_{ij} f_j$. A higher-dimensional analogue will use higher-dimensional indices for $i$ and $j$. So the goal of reshape([2, d / 2], iv) is to expose these two indices $i$ and $j$, belonging to the domain and codomain of $a$. As reshaping $A$ to [m1, ..., md, n1, ..., nd] would give us the indices in order $i_1, \cdots, i_d, j_1, \cdots, j_d$, we transpose the reshaped iv to end up with the desired order $i_1, j_1, \cdots, i_d, j_d$.

## 10.4 Runtime Evaluation

In the previous sections we demonstrated generic specifications of the blocked matrix multiplication. Now, we want to apply these algorithms on a particular architecture and demonstrate, explain how to chose blocking factors and compare our results with state of the art implementations: OpenBLAS [242], BLIS [234] and MKL [121].

### Experimental Setup

We use a multicore machine consisting of two 16 core AMD EPYC 7313 CPUs running at a 3.0 GHz frequency. The machine has private 32KiB L1D and 1MiB L2 caches. Every four cores share a 16 MiB L3 cache. We use OpenBLAS version 0.3.20, SaC 1.3.3 with GCC version 11.3.0, BLIS 0.9.0 and Intel MKL version 2020.4.304. The peak performance of this machine is 1536 Gflops/s.

Each core can do two 32 byte reads and one 32 byte write per clock cycle. That means that the L1 read bandwidth is $3 \cdot 2 \cdot 32 \cdot 32 = 6144$ GB/s. The L2 bandwidth is half this, 3072 GB/s. Every clock cycle a L2 cache can load in 32 bytes from L3, so the L3 bandwidth is $3 \cdot 32 \cdot (32/4) = 768$ GB/s. We have 8 times 16 GB of RAM, running at 3200 MT/s over 8 channels, resulting in 204.8 GB/s bandwidth between RAM and CPU.

**Bandwidth vs Compute Rate** The execution speed is limited by how fast we can do operations, and by how quickly we can get data from RAM into the CPU registers. For our machine the latter is:

$$\frac{204.8\text{GB/s}}{8\text{B} \cdot 3\text{Ghz}} \approx 8.5 \text{ doubles (8 bytes) per clock cycle}$$

Our machine can do $32{\cdot}4{\cdot}2{\cdot}2 = 512$ flops (cores, SIMD, fma, instructions per clock). So if we only use each double once, we can expect no more than $\frac{8.5}{512} \approx \frac{1}{60}$th of the peak performance. Therefore, in order to achieve peak performance we are forced to reuse each double we obtain from the memory multiple times. For example, if we reuse each double 30 times, we get half of the peak performance, etc.

In principle we have an intensity of $2mkn/(mk+kn)$ flops per double which tells us the problem is compute bound. Of course in practice it is hard to achieve this performance because we cannot store the entire matrix in registers. For this reason we look at the reuse of elements before they are evicted from their respective level in the memory hierarchy. So the unblocked implementation will have an effective intensity of less than $\frac{2\cdot k}{k} = 2$ as each calculation $(AB)_{ij} = \sum_{p=0}^{k} A_{ip}B_{pj}$ will see the $j$th column of $B$ being evicted from cache.

## Blocking Factors

In order to achieve peak performance, we use the approach from [91, 234] to find the block sizes that agree with the cache sizes for multiplying $A \in \mathbb{R}^{m\times k}$ by $B \in \mathbb{R}^{k\times n}$. When choosing block sizes we took measurements ranging from filling about half of the cache to the entire cache. We then picked the point where increasing the block size did not further improve runtimes. This way the inner blocks stay small, and we have some more leeway when choosing the outer blocks.

**First Blocking** We are going to chose our block sizes in a bottom-up fashion. We specify the size of a matrix as a superscript. Firstly, we notice that the matrix of size $6 \times 8$ can be stored entirely in SIMD registers. Our machine has 16 YMM registers, 32 bytes (4 doubles) each. Also, this leaves us at least three registers to do other operations. This means that for a multiplication of a $6 \times k$ and $k \times 8$ matrix, we store only 48 doubles on $2 \cdot 6 \cdot 8 \cdot k$ flops. For $A^{6\times k}$ and $B^{k\times 8}$ we get a 6-fold reuse of $B$ and an 8-fold reuse of $A$. Our processor can execute 2 fma instructions per clock cycle, and the L1, L2 cache can provide these operands in 2 and 4 cycles respectively. As 6 and 8 are larger than 4, it suffices to have our matrices in either L1 or L2. SHAPES SO FAR: $[6, 250]$ for A and $[250, 8]$ for B.

**Second Blocking** The second blocking level stacks all the $A^{6\times k}$ (denoted $A'$ here) into $A^{m\cdot 6\times k}$, performing the following operation:

$$A^{m\cdot 6\times k} = \begin{pmatrix} A'_1 \\ \vdots \\ A'_m \end{pmatrix} \qquad AB = \begin{pmatrix} A'_1 B \\ \vdots \\ A'_m B \end{pmatrix}.$$

We chose $m$ and $k$ such that $B$ fits into L1 and $A$ fits into L2. At the same time, we want to chose $m$ such that $B$ can be streamed into L1 (i.e. $m \cdot 6 > 60$) so that we can make the next blocking step. We chose $m = 15$ and $k = 250$. SHAPES SO FAR: $[15, 1, 6, 250]$ for $A$ and $[1, 1, 250, 8]$ for $B$.

**Third Blocking** After that, we consider stacking previously defined blocks $A^{90\times250}$ and $B^{250\times8}$ (denoted $A'$ and $B'$) into column and row vectors:

$$AB = \begin{pmatrix} A'_1 \\ \vdots \\ A'_k \end{pmatrix} \begin{pmatrix} B'_1 & \cdots & B'_n \end{pmatrix} \quad \text{or} \quad (AB)_{ij} = A'_i B'_j.$$

Each of these matrix-multiplications is a smaller matrix-multiplication described above, and this can be calculated at peak compute rate whenever $A'_i$ is in L2 cache and $B'_j$ can be streamed into L1 fast enough. As each element of $B'_j$ is reused $90 > 60$ times, this should be the case. However, we cannot guarantee that the entirety of $B'_{j+1}$ will be prefetched while we are calculating $A'_i B'_p$. Also, loading in a block of $A$ has some latency. So in practice we should to stack only as many blocks of $A$ and $B$ as we can fit into L3 cache in order to accelerate loading these blocks into L1 and L2 cache. We choose to stack 125 such blocks of $B$ and 3 of $A$. SHAPES SO FAR: $[3, 1, 15, 1, 6, 250]$ for $A$ and $[1, 125, 1, 1, 250, 8]$ for $B$.

**Remainder** For matrices larger than $A^{270\times250}$ and $B^{250\times1000}$, we compute a 'normal' matrix-multiplication in terms of these blocks (denoted $A'$ and $B'$):

$$AB = \begin{pmatrix} A'_{1,1} & \cdots & A'_{1,\frac{K}{250}} \\ \vdots & \ddots & \vdots \\ A'_{\frac{M}{270},1} & \cdots & A'_{\frac{M}{270},\frac{K}{250}} \end{pmatrix} \begin{pmatrix} B'_{1,1} & \cdots & B'_{1,\frac{N}{1000}} \\ \vdots & \ddots & \vdots \\ B'_{\frac{K}{250},1} & \cdots & B'_{\frac{K}{250},\frac{N}{1000}} \end{pmatrix}$$

FINAL SHAPES: $[\frac{m}{270}, \frac{k}{250}, 3, 1, 15, 1, 6, 250]$ for $A$ and $[\frac{k}{250}, \frac{n}{1000}, 1, 125, 1, 1, 250, 8]$ for $B$.

## Adjustments

Our initial experiments demonstrated the following two problems. Firstly, when SaC is making a recursive call with a submatrix, it does not detect read-only access to the sub-block, and makes a copy. Secondly, neither SaC, nor the underlying C compilers can generate the matrix multiplication kernel that operates in registers. Sadly as it can be, we have to introduce the following workarounds.

**Good Kernel** We use the FFI mechanism of SaC and implement the kernel using GCC vector extensions [222, 89]. This gives us a nice and easy formulation of the kernel (Listing 10.7) that is portable across all the architectures supported by GCC.

**Bad Memory** The SaC-compiler used in this chapter allocates and copies the subarrays in Listing 10.4. To avoid this, we pass the main array and the index into the block. The overloading of this version, Fig. 10.2 happens on the last two arguments that increase on each recursive step by two elements. We know that

```
// MR = 6; KC = 250; NR = 8
typedef double vect
  __attribute__((__vector_size__(32), aligned(8)));
void matmul(double **cp, double *a, double *b,
            int offset_a, int offset_b)
{
  double *c = malloc(MR * NR * sizeof(double));
  *cp = c; a = a + offset_a; b = b + offset_b;

  real (*as)[MR][KC   ] = (real (*)[MR][KC   ])a;
  vect (*bv)[KC][NR/4] = (vect (*)[KC][NR/4])b;
  vect (*cv)[MR][NR/4] = (vect (*)[MR][NR/4])c;

  vect zero = {0.,0.,0.,0.};
  vect c00 = zero, c01 = zero;
  vect c10 = zero, c11 = zero;
  vect c20 = zero, c21 = zero;
  vect c30 = zero, c31 = zero;
  vect c40 = zero, c41 = zero;
  vect c50 = zero, c51 = zero;
  for (size_t k = 0; k < KC; k++) {
    vect b0 = (*bv)[k][0];
    vect b1 = (*bv)[k][1];
    real a0 = (*as)[0][k];
    real a1 = (*as)[1][k];
    real a2 = (*as)[2][k];
    real a3 = (*as)[3][k];
    real a4 = (*as)[4][k];
    real a5 = (*as)[5][k];

    c00 += a0 * b0; c01 += a0 * b1;
    c10 += a1 * b0; c11 += a1 * b1;
    c20 += a2 * b0; c21 += a2 * b1;
    c30 += a3 * b0; c31 += a3 * b1;
    c40 += a4 * b0; c41 += a4 * b1;
    c50 += a5 * b0; c51 += a5 * b1;
  }
  (*cv)[0][0] = c00; (*cv)[0][1] = c01;
  (*cv)[1][0] = c10; (*cv)[1][1] = c11;
  (*cv)[2][0] = c20; (*cv)[2][1] = c21;
  (*cv)[3][0] = c30; (*cv)[3][1] = c31;
  (*cv)[4][0] = c40; (*cv)[4][1] = c41;
  (*cv)[5][0] = c50; (*cv)[5][1] = c51;
}
```

Listing 10.7: Handwritten kernel

we are dealing with 8-dimensional array, so the base case of our overloading calls our kernel. We also inline the rsum to avoid the creation of a temporary array.

```
double[m, n, di:shpci]
matmul(double[dout:shpao, m, k, di:shpai] A, int[dout] ia,
       double[dout:shpbo, k, n, di:shpbi] B, int[dout] ib)
{
  shpci = {[i] -> (i % 2 == 0) ? shpai[i] : shpbi[i]
                | [i] < [di]};
  return {[i, j] -> with { /* Inlined rsum */
                        ([0] <= [p] < [k]):
                          matmul(A, (ia ++ [i, p]),
                                 B, (ib ++ [p, j]));
                      }: fold(+, genarray(shpci, 0d))
          | [i, j] < [m, n]};
}

/* Base case for 4-levels of blocking */
double[m, n]
matmul(double[6:shpao, m, k] A, int[6] ia,
       double[6:shpbo, k, n] B, int[6] ib)
{
  return {[i, j] -> with {
                        ([0] <= [p] < [k]):
                          A[ia ++ [i, p]] *
                          B[ib ++ [p, j]];
                      }: fold(+, 0d)
          | [i, j] < [m, n]};
}
```

Figure 10.2: SaC workaround for copying out memory

This problem has been encountered before and studied in [211]. The authors believe it should be possible to extend this optimisation to be applicable to matrix-multiplication as well.

## Timings

We have measured a multiplication of a $8640 \times 10000$ matrix by a $10000 \times 10000$ matrix. We provide two SaC versions. The first one is blocked as described in Section 10.4, with a handwritten kernel for the two-dimensional computation. The second one does not use a handwritten kernel. As GCC cannot use blocking at the register level, we use squarish blocks: $40 \times 40$ blocks to target L1, and $240 \times 200$ for $A, C$ and $200 \times 200$ for $B$ to target L2. We compare it to the state of the art implementation OpenBLAS. The timings are averaged over 10 runs, and we draw error bars from the slowest to the fastest run. These results

are summarised on the left-hand side of Fig. 10.3. The axes are log-scaled, so linear speedup is also linear in the graph. The fastest programs are rather close to each other, so we also graph the efficiencies for these (on the right-hand side of Fig. 10.3) compared to the sequential performance to OpenBLAS. The astute reader may have noticed that OpenBLAS and MKL perform better than the 48 Gflops/s theoretical peak performance on one core. It could be due to Strassen's algorithm as explained in [116], but we have not found a confirmation of this in the literature or OpenBLAS's codebase. The alternative is that the actual clock speeds are higher than 3 GHz, even though we have verified that the settings cap it at 3 GHz.

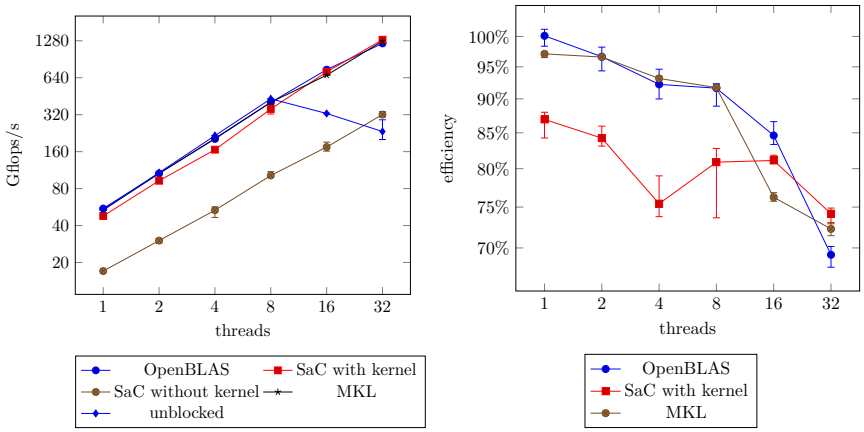



Figure 10.3: Performance of computing $AB$ for matrix $A$ of size $8640 \times 10000$ and $B$ of size $10000 \times 10000$. On the left-hand side the absolute figures are presented, on the right hand side we show the efficiencies (per core) compared to OpenBLAS's sequential performance. The SaC version uses four-level blocking described above.

OpenBLAS achieves 55 Gflops/s on one core, MKL achieves 53 Gflops/s and SaC with kernel 48 Gflops/s. A reason that the SaC kernel is slower, is that it allocates memory for the $6 \times 8$ block of $C$ instead of directly writing to the memory of $C$. MKL and OpenBLAS, opposed to the SaC version, does not have blocks stored in contiguous memory, so they need to pack these blocks. We see that this causes the SaC version to scale better and catch up to OpenBLAS higher core counts: 1305 Gflops/s on 32 cores for SaC, 1218 Gflops/s for OpenBLAS, and 1272 Gflops/s for MKL. The reason this only happens at higher core counts is that packing takes a little bandwidth, which is more contested the more cores are used. SaC in contrast does not have to pack, as we have benchmarked the preblocked version. The SaC version without a kernel is about a factor four slower, and has less speedup going from 16 to 32 cores. On 32 cores, compared to the fastest sequential time by OpenBLAS, we have efficiencies of 69%, 72%, 74%, 18% for OpenBLAS, MKL, SaC with kernel, SaC without kernel, respectively. The unblocked version is unsurprisingly very slow, 12 Gflops/s on 32 cores, a factor 100 slower than SaC with kernel, and 27 times slower than the blocked SaC version without kernel. We

reuse each element once before it is evicted from cache, so we cannot expect more than $\frac{1532}{60} \approx 26$ Gflops/s. Combined with the strided access of $B$, it gives the poor result.

For BLIS we use use OpenMP for the backend and thread-binding strategy OMP_PROC_BIND=close and
OMP_PLACES=cores as recommended by the authors. It does well up to including 8 cores, and then experiences slowdown.

### Blocking and Unblocking

The SaC compiler is not required to materialize $A$, $B$ in memory, block them, and then run the matrix-multiplication. Instead, it could immediately generate $A$, $B$ in blocked form. For this reason we did not include the blocking cost in Fig. 10.3. In the following table we list those times for cases where the SaC compiler cannot immediately generate $A, B$ in blocked form. As before we use $A^{8640\times10^4}$, $B^{10^4\times10^4}$ splitting the runtime between blocking, matrix-multiplication, and unblocking in seconds, averaging the time over 10 runs.

| Proc | Block | $\sigma$ | Matmul | $\sigma$ | Unblock | $\sigma$ |
|---|---|---|---|---|---|---|
| 1 | 5.317 | 0.118 | 38.78 | 0.19 | 2.799 | 0.081 |
| 2 | 2.675 | 0.011 | 19.60 | 0.06 | 1.494 | 0.010 |
| 4 | 1.439 | 0.208 | 9.98 | 0.05 | 0.770 | 0.022 |
| 8 | 0.714 | 0.044 | 5.00 | 0.02 | 0.417 | 0.055 |
| 16 | 0.464 | 0.009 | 2.53 | 0.01 | 0.246 | 0.022 |
| 32 | 0.302 | 0.007 | 1.44 | 0.01 | 0.124 | 0.029 |

## 10.5 Related Work

The idea to use blocking to improve performance of numerical algorithms dates back to the 1960s [170], roughly coinciding with the introduction of memory hierarchies. Since then, a large body of research has investigated [149, 138] ways on how to implement blocked linear algebra operations efficiently. This work provides guidelines on how to structure blocked algorithms, assuming manual implementations. The canonical results in this area are described in [91] which constitutes the basis for today's state of the art hand-written libraries such as OpenBLAS [242] and BLIS [234]. In contrast to our work, the manual encodings in low-level languages of these approaches raises potential concerns about their correctness and their maintainability.

Another area of research in the context of blocking aims to enable compilers to introduce blocking into blocking-free implementations. This work is based on dependency analyses such as the ones in [5] and the polyhedral model [87] which provides a mathematical framework for analysing and transforming loop nests. More recent work in this context [83] includes possible data layout transformations as well, which, at least in principle, enables compiler-driven transformations into code very similar to the code that is generated from our shape-generic specification.

One of the key challenges of the compiler-introduced blocking approach is the need to find suitable parameters. Analytical models have been proposed [160], but

there are also attempts to tune the parameters based on exhaustive experiments or by applying some form of machine learning. ATLAS [243] is an implementation of BLAS that automatically tunes architecture specific parameters. Further examples of autotuning can be found in [248] and [229]. Lift [225] uses autotuning, by applying rewrite rules to a high-level functional specification of the algorithm. While our approach does not solve the parameter finding problem either, it is possible for the programmer to experiment with different blocking scenarios without rewriting the rank-polymorphic matrix multiply at all. This is similar in spirit to approaches such as those of Halide [201] or FLAME [246], which offer the programmer to specify separately which transformations should be introduced by the compiler. Here, the key difference is that our approach allows the blocking to be encoded in the array shapes and, thus, in the source language itself.

Finally, this work relates to earlier work on shape-guided parallel implementations of scan operations described in [221]. The difference is that here we use the shape to guide the blocking instead of the parallelism.

## 10.6 Conclusion

This chapter proposes to implement blocked numerical algorithms through rank-polymorphic functions on arrays. This allows for a concise formulation of arbitrary blocking levels, enabling programmers to encode the desired blocking factors through array shapes.

When striving for the highest levels of performance, we see that some further optimisations within the SaC compiler would be desirable. Specifically, the ability to implement sub-array selections without copying elements seems crucial in this context. While this mainly constitutes an engineering challenge, the chapter shows that we can achieve these effects by means of a workaround in the form of a re-write and a 40-line kernel written in C. Jointly, these tweaks lead to parallel performance that is competitive with state-of-the-art libraries BLIS, OpenBLAS and MKL.

### Further work

The current implementation of the SaC compiler parallelises over the outer-dimension. When the outer-dimension is small, this limits scalability and can cause issues in the load-balancing. These problems would be mitigated by parallelising over more dimensions.

When the blocking sizes do not divide the matrix sizes, a padding scheme must be used.

Finally, rank polymorphic arrays are a natural fit for tensors. Tensor contractions are an operation very similar to matrix multiplication, so it would be interesting to see this framework applied to them.

# Chapter 11

# Quickhull is Usually Forward Stable

## 11.1 Introduction

The convex hull $CH(P)$ of a bounded set $P \subseteq \mathbb{R}^2$ is the smallest set that contains $P$ and is closed under taking convex combinations. On a computer, we can only deal with finite sets $P$ and the convex hull of a finite set is always a polygon. The planar convex hull problem is to list the vertices of this polygon in clockwise (or counter-clockwise) order. By abuse of notation, we will write $|CH(P)|$ for the number of vertices on the convex hull.

The goal in algorithm design is to compute a good solution quickly. There are many convex hull algorithms to choose from, differing in worst-case runtime complexity and practical performance. The Quickhull algorithm is most performant on many data sets, usually running in $O(|P| \log |CH(P)|)$, despite a worst-case complexity of $O(|P|^2)$. The goal is to compute a good solution and not **the** solution because the convex hull is defined on values in $\mathbb{R}$. This introduces uncertainties arising from measurements and rounding errors. Dealing with these uncertainties is a well-established field in mathematics, but has not been applied to Quickhull yet. This paper seeks to close that gap.

Using some jargon—which we explain assuming no prior knowledge in Subsection 11.2—this paper provides the following contributions.

- We define a measure of inaccuracy for the convex hull problem and do the corresponding perturbation analysis.
- We show that the Quickhull algorithm has a forward error bound of $O(|P|^2\epsilon)$ for $\epsilon$ the machine precision.
- We give a probabilistic argument that in practice the error bound is more likely to grow proportionally to $\log(|CH(P)|)\epsilon$.

- We propose two variations of Quickhull that reduce the worst-case error bound to $O(\sqrt{|P|}\log(|P|)\epsilon)$ or $O((\log|P|)^2\epsilon)$ for cases where the runtime of Quickhull is $O(|P|\log|P|)$.

## 11.2 Background

### Quickhull

The Quickhull algorithm [37] makes use of two geometric observations illustrated by Figure 11.1. First, if $p$, $q$ are in the convex hull, then any point $r$ with maximum distance to the line through $p$ and $q$ is in the convex hull as well. Second, any point within the triangle $\Delta prq$ is not in the convex hull. So if the points $prq$ are listed in clockwise order, we only need to consider the points to the left of the oriented line segment $pr$ and to the left of the oriented line segment $rq$. This idea can be applied recursively to obtain Algorithm 20.

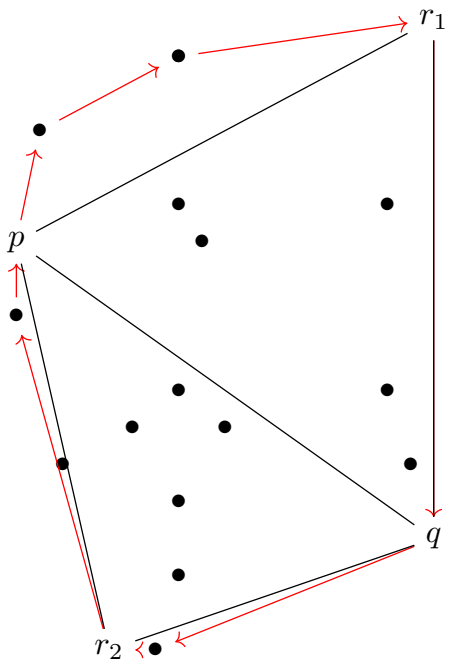



Figure 11.1: The points $p$ and $q$ are the left-most and right-most points, and therefore in the hull. The points $r_1$, $r_2$ are farthest from the line $pq$, and also in the hull. Any point within the two triangles $\Delta pr_1q$, $\Delta qr_2p$ cannot be in the convex hull.

While it is intuitively clear what points are to the left of a line $pq$, or inside a triangle $prq$, we need a quantity called the orientation of three points to define

**Algorithm 20** Quickhull algorithm.

**Input:** $P$ : set of points in $\mathbb{R}^2$.
**Output:** $H$ : vertices of convex hull of $P$ in clockwise order.

Let $p$ be the left-most point
Let $q$ the right-most point
$S_1 := \{u \in P \mid u \text{ to the left of } pq\}$
$r_1 := \operatorname{argmax}_{u \in S_1} d(u, pq)$.
$S_2 := \{u \in P \mid u \text{ to the right of } pq\}$
$r_2 := \operatorname{argmax}_{u \in S_2} d(u, pq)$.
$H := \{p\} \cup \textsc{Hull}(S_1, p, r_1, q) \cup \{q\} \cup \textsc{Hull}(S_2, q, r_2, p)$.
**function** HULL(P, p, r, q)
    **if** $|P| \leq 1$ **then**
        $H := P$.
    **else**
        $S_1 := \{u \in P \mid u \text{ to the left of } pr\}$
        $r_1 := \operatorname{argmax}_{u \in S_1} d(u, pr)$.
        $S_2 := \{u \in P \mid u \text{ to the left of } rq\}$
        $r_2 := \operatorname{argmax}_{u \in S_2} d(u, rq)$.
        $H := \{p\} \cup \textsc{Hull}(S_1, p, r_1, r) \cup \{r\} \cup \textsc{Hull}(S_2, r, r_2, q) \cup \{q\}$.

this precisely. Given three points $p, u, q$ in the plane, we have vectors

$$\vec{up} = \begin{pmatrix} p_x - u_x \\ p_y - u_y \end{pmatrix}, \vec{uq} = \begin{pmatrix} q_x - u_x \\ q_y - u_y \end{pmatrix}$$

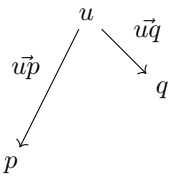
representing oriented line segments between the points, as depicted in Figure 11.2a.

We can view these as vectors in $\mathbb{R}^3$ by adding a $z$-coordinate 0. Now the cross product $\vec{up} \times \vec{uq}$ is equal to

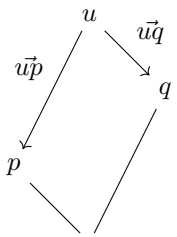
$$\begin{pmatrix} 0 \\ 0 \\ (p_x - u_x) \cdot (q_y - u_y) - (p_y - u_y) \cdot (q_x - u_x) \end{pmatrix}.$$

We write $\text{orient}(p, u, q)$ for the $z$-coordinate $(p_x - u_x) \cdot (q_y - u_y) - (p_y - u_y) \cdot (q_x - u_x)$ If the cross-product points up, so if $\text{orient}(p, u, q) > 0$, we make a right turn going from $p$ to $u$ to $q$, and if this is negative, we make a left turn. If it is zero, the points $p, u, q$ are collinear. This gives us the precise definition of what it means for $u$ to be to the left of $pq$: $\text{orient}(p, u, q) > 0$.

We can also use the orientation to find the point farthest from the line segment $pq$ [109]. The norm of the outer product $|\vec{up} \times \vec{uq}| = |\text{orient}(p, u, q)|$ is the area of the parallelogram spanned by $\vec{up}$ and $\vec{uq}$. This is also equal to the length of $pq$ multiplied by the distance from $u$ to the line through $pq$. So $|\text{orient}(p, u, q)| > |\text{orient}(p, u', q)|$ if and only if $u$ is farther from $pq$ than $u'$.




(a) Two vectors induced by three points in the plane. The path $p, u, q$ makes a right turn.




(b) Parallelogram spanned by $\vec{up}$ and $\vec{uq}$. The area of this parallelogram is $|\operatorname{orient}(p, u, q)|$.

Figure 11.2: Vectors between points in the plane.

## Numerical Robustness

It is unfeasible to compute exact solutions to real-valued functions because the input is imprecise. This imprecision can arise from the rounding of values in $\mathbb{R}$ to a finite data type `double` or `float`, but also from imprecision in measuring physical quantities, or truncation errors in approximating continuous functions with discrete versions.

Instead of aiming to compute the correct result, we aim to find bounds on the error. The error between the true solution $f(x)$ of a problem $x$, and the computed solution $\widehat{f(x)}$ is called the *forward error*. We may also ask ourselves: for what $\tilde{x}$ is our computed solution $\widehat{f(x)}$ the actual solution $f(\tilde{x})$? We call the error between $x$ and $\tilde{x}$ the *backward error*. If the backward error is smaller than the uncertainty in the input, the solution is as good as we can expect.

We can connect these two ideas together with a *perturbation analysis*: how is the error between $x$ and $\tilde{x}$ in the input magnified to an error in the output between $f(x)$ and $f(\tilde{x})$? If the error in the output has the same or smaller magnitude we call the problem *well-conditioned*, if the error in the output is substantially larger we call the problem *ill-conditioned*. For a well-conditioned problem, a small backward error also implies a small forward error.

Algorithms whose forward or backward error grows slowly with respect to the problem size are called *numerically stable*.

**Example 3.** *Subtraction is a classic example, and will also be useful for our analysis. For non-zero numbers in $\mathbb{R}$, we define the relative error of $\hat{x}$ approximating $x$ as the quantity $|x - \hat{x}|/|x|$, or equivalently $|\delta|$ for $\hat{x} = (1 + \delta)x$. The computed difference $fl(a-b)$ of floating point numbers $a$, $b$ satisfies $fl(a-b) = (1+\delta)(a-b)$, where $\delta$ is smaller than the machine precision $\epsilon$, $2^{-24}$ for single-precision and $2^{-53}$ for double-precision. This means we have a forward error bounded by $\epsilon$, and by rewriting to $fl(a - b) = (1 + \epsilon)a - (1 + \epsilon)b$, also the backward error is bounded by $\epsilon$. For the perturbation analysis, we subtract two perturbed values $\hat{a} = (1 + \delta_a)a$*

*and* $\hat{b} = (1 + \delta_b)b$ *and compute the relative error*

$$\left| \frac{(a-b) - (\hat{a} - \hat{b})}{(a-b)} \right| = \frac{|\delta_a a + \delta_b b|}{|a-b|} \leq \max(|\delta_a|, |\delta_b|) \frac{|a| + |b|}{|a-b|}.$$

*We see that subtraction is ill-conditioned when* $a \approx b$*: small errors* $\delta_a$*,* $\delta_b$ *in the input are magnified by the subtraction. The reader may recognize this as catastrophic cancellation. Note that this is only problematic if* $a$*,* $b$ *have been contaminated by error.*

## 11.3 Perturbation Analysis

To bound an error, we first need to define how far a computed solution is from the true solution. As we work with geometric shapes, it is intuitive to define error in terms of distance. The standard metric is the Hausdorff distance $d(A, B) := \max\left(\max_{a \in A} d(a, B), \max_{b \in B} d(A, b)\right)$. This has as unfortunate property that the distance is not invariant to scaling: we have $d(\{(1.00, 1.00)\}, \{(2.00, 2.00)\}) = \sqrt{2}$, but $d(\{(100, 100)\}, \{(200, 200)\}) = 100\sqrt{2}$, while these may very well be the same data, one expressed in metres, and the other in centimetres. If we have a reference set—in our case the input—we can divide the Hausdorff distance by the maximum absolute value of $x$- and $y$-coordinates $M$, yielding $d_M(A, B) := d(A, B)/M$. This makes the measure of error independent of the chosen unit. The resulting function $d_M$ satisfies the axioms of a distance.

We can now do the perturbation analysis with respect to this error. We write $\delta = d_M(\tilde{P}, P)$ for the error between $P$ and perturbed set $\tilde{P}$. To bound the Hausdorff distance we take a point $a \in CH(P)$. We can write this as a convex combination $a = \sum_{i=1}^{n} \lambda_i a_i$ for $a_i \in P$ and $\sum_{i=1}^{n} \lambda_i = 1$, $\lambda_i \geq 0$. By definition of the Hausdorff distance, there exist $b_i \in \tilde{P}$ such that $d(a_i, b_i) \leq M\delta$. This gives a point $b = \sum_{i=1}^{n} \lambda_i b_i$ that is on the convex hull of $\tilde{P}$, and close to $a$. The triangle inequality bounds the distance

$$d(a, b) = |a - b| \leq \sum_{i=1}^{n} \lambda_i |a_i - b_i| \leq \sum_{i=1}^{n} \lambda_i M\delta = \delta M.$$

Therefore $d(CH(P), CH(\tilde{P})) \leq \delta M$, so $d_M(CH(P), CH(\tilde{P})) \leq \delta = d_M(P, \tilde{P})$. As the backward error leads to a forward error of the same magnitude, we can consider the convex hull problem well-conditioned.

## 11.4 Geometric Tests

As preparation for the full analysis, we first consider the steps of Algorithm 20 in isolation. Recall from the background section that we can decide whether $puq$ makes a right turn by testing $\text{orient}(p, u, q) > 0$. To reduce the number of floating point operations necessary to evaluate this in a loop over $u$, we rewrite the inequality to $(p_x - u_x) \cdot (q_y - p_y) > (p_y - u_y) \cdot (q_x - p_x)$. The subtractions

$q_y - p_y$ and $q_x - p_x$ can be computed once and reused every iteration. We denote the true evaluation of this inequality by $\mathrm{rt}(p, u, q)$ and the computed evaluation by $\widehat{\mathrm{rt}}(p, u, q)$.

We can test whether $u$ is farther from $pq$ than $u'$ by comparing $\mathrm{orient}(p, u, q)$ to $\mathrm{orient}(p, u', q)$. This makes it tempting to compute the orientation once for each point, and then do both the right-turn test and the distance test with this quantity. However, this can run into precision problems because the middle subtraction in $(u_x - p_x)(q_y - u_y) - (u_y - p_y)(q_x - u_x)$ takes arguments that are polluted by error and this might incur numerical cancellation (Example 3). Instead, we rewrite the inequality

$$\begin{aligned}(p_x - u_x)(q_y - u_y) - (p_y - u_y)(q_x - u_x) <\\ (p'_x - u_x)(q_y - u'_y) - (p'_y - u_y)(q_x - u'_x)\end{aligned}$$

to

$$(q_y - p_y)(u_x - u'_x) < (q_x - p_x)(u_y - u'_y).$$

This eliminates any subtractions on computed values. We write $\mathrm{frt}(p, q, u, u')$ for the predicate $d(pq, u) > d(pq, u')$, and $\widehat{\mathrm{frt}}(p, q, u, u')$ for the computed predicate.

Finding the point farthest from $pq$ is a reduction

$$u_1 \operatorname{op} \cdots \operatorname{op} u_2$$

where $u_1 \operatorname{op} u_2$ returns $u_1$ if $\mathrm{frt}(p, q, u_1, u_2)$ and $u_2$ otherwise. Mathematically this gives a point with maximum distance to $pq$ regardless of how we order the operations. However, this is not true for the floating point evaluation $\widehat{\mathrm{frt}}$. We write this reduction as $\operatorname{reduce}\widehat{\mathrm{frt}}(p, q, -, -)X$ for some choice evaluation order over points in $X$. We will analyse some choices in Subsection 11.4.

Taken together, this gives Algorithm 21.

## Right Turn

Lemma 2 shows $\widehat{rt}(p, u, q)$ is accurate for points far from $pq$. This is a desirable property because we only misclassify points that have little effect on result.

**Lemma 2.** *If* $\mathrm{rt}(p, u, q)$ *is true, but* $\widehat{\mathrm{rt}}(p, u, q)$ *is not, or the other way around, then* $d(u, pq) \leq \gamma_6 M$*, where* $\gamma_6 = \frac{6\epsilon}{1-6\epsilon}$ *for* $\epsilon$ *the machine precision.*

*Proof.* As $\mathrm{orient}(p, u, q)$ is equal to the parallelogram spanned by $\vec{up}$ and $\vec{uq}$, it is equal to twice the area of triangle $puq$. This gives an upper bound

$$d(u, pq) \leq \frac{|\operatorname{orient}(p, u, q)|}{2\|p - q\|},$$

so we start by showing that a conflict between $\widehat{\mathrm{rt}}$ and rt implies small $|\operatorname{orient}(p, u, q)|$.

**Algorithm 21** Floating point Quickhull algorithm

**Input:** $P$ : set of points in $\mathbb{R}^2$.
**Output:** $H$ : vertices of convex hull of $P$ in clockwise order.

```
1: Let p be the left-most point
2: Let q the right-most point
3: S_1 := {u ∈ P | r̂t(p, u, q)}.
4: r_1 := reduce f̂rt(p, q, −, −)S_1.
5: S_2 := {u ∈ P | r̂t(q, u, p)}.
6: r_2 := reduce f̂rt(q, p, −, −)S_2.
7: H := {p}∪ Hull(S_1, p, r_1, q) ∪{q}∪ Hull(S_2, q, r_2, p).
8: function Hull(P, p, r, q)
9:     if |P| ≤ 1 then
10:        H := P.
11:    else
12:        S_1 := {u ∈ P | r̂t(p, u, r)}.
13:        r_1 := reduce f̂rt(p, r, −, −)S_1.
14:        S_2 := {u ∈ P | r̂t(r, u, q)}
15:        r_2 := reduce f̂rt(r, q, −, −)S_2.
16:        H := {p}∪ Hull(S_1, p, r_1, r) ∪{r}∪ Hull(S_2, r, r_2, q) ∪{q}.
```

Suppose $\mathrm{rt}(p,u,q)$ is true, but $\widehat{\mathrm{rt}}(p,u,q)$ is false. We write $fl(\cdots)$ for the floating point evaluation of a primitive operation. As the IEEE-754 standard satisfies $fl(x \circ y) = (1+\delta)(x \circ y)$, $|\delta| < \epsilon$ for $\circ \in \{+,-,\cdot\}$, the test

$$\widehat{\mathrm{rt}}(p,u,q) \equiv fl(fl(p_x - u_x)fl(q_y - p_y)) < fl(fl(p_y - u_y)fl(q_x - p_x))$$

implies

$$(p_x - u_x)(q_y - p_y) < \frac{(1+\delta_1)(1+\delta_2)(1+\delta_3)}{(1+\delta_4)(1+\delta_5)(1+\delta_6)}(p_y - u_y)(q_x - p_x). \tag{11.1}$$

We can simplify this inequality by noting there exists a $\theta$ such that

$$\frac{(1+\delta_1)(1+\delta_2)(1+\delta_3)}{(1+\delta_4)(1+\delta_5)(1+\delta_6)} = 1 + \theta$$

and $|\theta| \leq \gamma_6$ [107, Lemma 3.1]. That yields

$$\begin{aligned}(p_x - u_x)(q_y - p_y) < (1+\theta)(p_y - u_y)(q_x - p_x) \equiv \\ -\theta(p_y - u_y)(q_x - p_x) < \mathrm{orient}(p,u,q).\end{aligned}$$

As $\mathrm{rt}(p,u,q)$ is false, we have $\mathrm{orient}(p,u,q) \leq 0$, so $|\,\mathrm{orient}(p,u,q)| \leq \gamma_6|(p_y - u_y)(q_x - p_x)|$. An analogous argument shows this also holds for $\mathrm{rt}(p,u,q)$ true but $\hat{\mathrm{rt}}(p,u,q)$ false.

We can now compute an upper bound

$$\begin{aligned} d(u, pq) &= \frac{|\operatorname{orient}(p, u, q)|}{2\|p - q\|} \\ &\leq \frac{\gamma_6 |(p_y - u_y)(q_x - p_x)|}{2\sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}} \\ &\leq \frac{1}{2}\gamma_6 |p_y - u_y| \\ &\leq \frac{1}{2}\gamma_6 (|p_y| + |u_y|) \\ &\leq \gamma_6 M. \end{aligned}$$

□

## Distance Test

Lemma 3 shows that at least the building blocks of the reductions—individual tests $\widehat{\text{frt}}$—are accurate.

**Lemma 3.** *If* $\widehat{\text{frt}}(p, q, u, u')$ *is true, but* $\text{frt}(p, q, u, u')$ *is false, then* $0 \leq d(u', pq) - d(u, pq) \leq \gamma_6 M$.

*Proof.* Similarly to Lemma 2 we assume $\widehat{\text{frt}}(p, q, u, u')$ true, but $\text{frt}(p, q, u, u')$ false. This gives the two inequalities

$$\begin{aligned} (q_y - p_y)(u_x - u'_x) - (q_x - p_x)(u_y - u'_y) &< \theta(q_x - p_x)(u_y - u'_y), \\ (q_y - p_y)(u_x - u'_x) - (q_x - p_x)(u_y - u'_y) &\geq 0, \end{aligned}$$

for some $|\theta| \leq \gamma_6$, which we can combine to

$$|(q_y - p_y)(u_x - u'_x) - (q_x - p_x)(u_y - u'_y)| \leq |\theta(q_x - p_x)(u_y - u'_y)|. \tag{11.2}$$

Looking at the left side from a geometric point of view gives

$$\begin{aligned} (q_y - p_y)(u_x - u'_x) - (q_x - p_x)(u_y - u'_y) &= \operatorname{orient}(p, u, q) - \operatorname{orient}(p, u', q) \\ &= 2d(u, pq)\|p - q\| - 2d(u', pq)\|p - q\| \\ &= 2\|p - q\|(d(u, pq) - d(u', pq)). \end{aligned} \tag{11.3}$$

Combining the bound of Equation 11.2 with the geometric view of Equation 11.3 gives

$$\begin{aligned} d(u, pq) - d(u', pq) &\leq |\theta| \frac{|(q_x - p_x)(u_y - u'_y)|}{2\|p - q\|} \\ &\leq \frac{1}{2}|\theta||u_y - u'_y| \leq |\theta| M. \end{aligned}$$

□

The accuracy combining these $\widehat{\text{frt}}$ tests depends on the evaluation order. We analyse three orders inspired by summation. The three algorithms 22, 23, 24 resemble normal summation, blocked summation, and pairwise summation respectively. Lemma 4 gives bounds for these algorithms.

**Algorithm 22** Simple way of finding farthest point from $pq$.

**Input:** $P$ : ordered set of points in $\mathbb{R}^2$, $p, q \in \mathbb{R}^2$.
**Output:** $u_{\max} \in P$ farthest from $pq$ with positive orient$(p, r, q)$.

```
1: u_max = P[0];
2: for i = 1 to |P| − 1 do
3:     if frt^(p, q, P[i], u_max) then
4:         u_max = P[i];
```

**Algorithm 23** Finding farthest point from $pq$ with blocking.

**Input:** $P$ : ordered set of points in $\mathbb{R}^2$, $p, q \in \mathbb{R}^2$.
**Output:** $u_{\max} \in P$ farthest from $pq$ with positive orient$(p, r, q)$.

```
1: u_max = P[0];
2: for I = 1 to |P| − 1 step m do
3:     b_max = P[I];
4:     for i = I + 1 to min(I + m − 1, |P| − 1) do
5:         if frt^(p, q, P[i], b_max) then
6:             b_max = P[i];
7:     if frt^(p, q, b_max, u_max) then
8:         u_max = b_max
```

**Lemma 4.** *Let $r$ be the farthest point from $pq$ out of $n$ points and $\widehat{r}$ a computed estimate of this point. We write*

$$F_n := \frac{d(r, pq) - d(\widehat{r}, pq)}{M}.$$

*for the scaled error. Algorithms 22, 23, 24 have asymptotic error bounds $O(n\epsilon)$, $O((m + n/m)\epsilon)$, $O(\log(n)\epsilon)$ on $F_n$ respectively.*

*Proof.* We first show how the error propagates through chains of incorrect decisions. We then prove the three cases by bounding the length of such a chain. Suppose $\widehat{\text{frt}}(p, q, u, u')$ is evaluated correctly. Then $d(u, pq) - d(u', pq) > 0$, so $d(r, pq) - d(u, pq) < d(r, pq) - d(u', pq)$, implying $u$ is a better guess than $u'$. If $\widehat{\text{frt}}(p, q, u, u')$ is evaluated incorrectly, then Lemma 3 shows $d(u', pq) - d(u, pq) < \gamma_6 M$, or $d(r, pq) - d(u, pq) < d(r, pq) - d(u', pq) + \gamma_6 M$. So $u'$ makes the guess at most $\gamma_6 M$ worse.

The first bound follows directly from doing at most $n$ comparisons and $\gamma_6 \in O(\epsilon)$. If we do this within a group of size $m$, we may end up with an error of $O(m\epsilon)$. We then compare the farthest point within each of these $n/m$ groups, so

**Algorithm 24** Finding farthest point from $pq$ with recursion.

**Input:** $P$ : ordered set of points in $\mathbb{R}^2$, $p, q \in \mathbb{R}^2$.
**Output:** $u_{\max} \in P$ farthest from $pq$ with positive orient$(p, r, q)$.

```
1: function FARTHEST(P, p, q)
2:     if |P| = 1 then
3:         u_max = P[0].
4:     else
5:         u^1_max = FARTHEST(P(1 : |P|/2 − 1), p, q).
6:         u^2_max = FARTHEST(P(|P|/2 : |P| − 1), p, q).
7:         if frt(p, q, u^1_max, u^2_max) then
8:             u_max = u^1_max.
9:         else
10:            u_max = u^2_max.
```

the final error may be $O((m+n/m)\epsilon)$. The final bound follows from the recursion having depth $\lceil \log_2 n \rceil$ and an induction. □

Example 4 shows we can get close to the worst-case bound $O(n\epsilon)$ even when assuming the rounding is random, as long as we can construct an adversarial input.

**Example 4.** *Consider the points in Figure 11.3 where $d(u_i, pq) - d(u_{i+1}, pq) \approx \gamma_6 M/k$ for some fixed $k$. We will keep updating our guess of the farthest point unless we evaluate $\widehat{\text{frt}}$ correctly $k$ times in a row. The probability of this happening is exponential in $k$, so we do not have to choose $k$ large relative to $n$ to likely end up with a point $u_i$ close to $u_n$. This point satisfies*

$$\frac{d(u_1, pq) - d(u_i, pq)}{M} \in O(n\epsilon).$$

Though the structure of the points is not especially adversarial, the order of points is. We hypothesize that even Algorithm 22 will be close to optimal, if the order of points is random.

**Hypothesis 1.** *Under random permutations, Algorithm 22 gives $\mathbb{E}[F_{|P|}] \in O(\epsilon)$.*

A further investigation of $F_{|P|}$ will show this hypothesis is related to the expected length of subsequences, making it a more complicated variation of Ulam's Problem [203] and therefore outside the scope of this paper. Instead, we will support Hypothesis 1 with a Monte Carlo method.

Let $u_1, \cdots, u_{|P|}$ be ordered by distance from $pq$. We write $u \sim u'$ for the relation $|d(u, pq) - d(u', pq)|/M < \gamma_6$, e.g. we cannot tell at our precision whether $u$ or $u'$ is farther from $pq$. If we make the wrong decision every time we evaluate $\widehat{frt}(u, u', p, q)$, a permutation $\sigma$ will end up with $u_{\sigma(|P|-m)}$ for the largest subsequence $u_{\sigma(|P|)} \sim u_{\sigma(|P|-1)} \sim \cdots \sim u_{\sigma(|P|-m)}$. We have an upper bound of $m\gamma_6$ on $F_{|P|}$.

The more points that are pairwise related, the larger the expected length of such a subsequence. However, for such a subsequence $u_{i+k-1}, \cdots, u_i$ satisfying

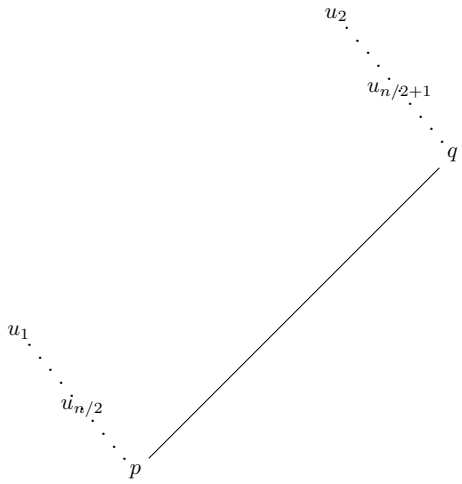



Figure 11.3: Adversarial input

$u_{i+k-1} \sim u_i$, each wrong guess in a subsequence only increases $F_n$ by $\gamma_6/k$ on average. To capture this effect, we create subsequences for different values of $k$, similar to the construction of Example 4. We let $|P|$ range from 256 to $2^{20}$, $k$ from 1 to 10, take 300 samples and repeat each experiment 10 times, reporting error bars in Figure 11.4. We see that $\mathbb{E}[F_{|P|}]$ appears to be constant in $k$ and $|P|$. The code for reproducing this experiment is available at [140].

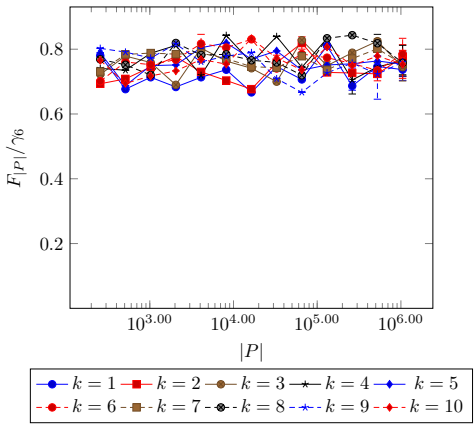



Figure 11.4: Monte-Carlo simulation of $F_n$, using $\gamma_6$ as unit.

## 11.5 Numerical Stability of Quickhull

Now that we understand the accuracy of the building blocks of Algorithm 21, we are ready to find bounds on the forward error. We can simplify the analysis by making the following observation. The first step of the convex hull algorithm finds the left-most and right-most points by using coordinate comparison, which is exact. After that the structure is the same as `HULL`. For this reason we restrict the analysis to the recursive function `HULL`.

**Theorem 5.** *The forward error of Quickhull is bounded by $2DF_{|P|}$, where $D$ is the depth of the recursion and $F_{|P|}$ is a bound of Lemma 4.*

**Proof sketch 1.** *We fix each point where $\widehat{rt}$ or $\widehat{frt}$ made an incorrect decision by moving them a little bit, using Lemma 2 and Lemma 4. This yields $\tilde{P} \approx P$, $\tilde{S}_1 \approx S_1$, and $\tilde{S}_2 \approx S_2$ such that $CH(\tilde{P} \cup \{p, q\}) = \{p\} \cup CH(\tilde{S}_1) \cup \{r\} \cup CH(\tilde{S}_2) \cup \{q\}$. We can push this backward error $d_M(P, \tilde{P})$ forward using the perturbation analysis. Combining this with an inductive argument on the recursion yields the following chain of approximations.*

$$
\begin{aligned}
\hat{H} &= \{p\} \cup \hat{H}_1 \cup \{r\} \cup \hat{H}_2 \cup \{q\} \\
&\quad \updownarrow \approx \text{ (induction hypothesis)} \\
&\{p\} \cup CH(S_1) \cup \{r\} \cup CH(S_2) \cup \{q\} \\
&\quad \updownarrow \approx \text{ (perturbation analysis)} \\
&\{p\} \cup CH(\tilde{S}_1) \cup \{r\} \cup CH(\tilde{S}_2) \cup \{q\} \\
&\quad \updownarrow = \text{ (construction of } \tilde{P}) \\
&CH(\tilde{P} \cup \{p, q\}) \\
&\quad \updownarrow \approx \text{ (perturbation analysis)} \\
&CH(P \cup \{p, q\})
\end{aligned}
$$

We can make the $\approx$ rigorous with triangle inequalities, which yields the following proof.

*Proof.* We construct $\tilde{P}$ as follows. By Lemma 2, all points that are classified incorrectly have a distance of at most $\gamma_6 M$ to either $pr$ or $rq$. If $u \in P$ has been incorrectly added $S_1$, we move it $\frac{1}{2}(d(u, pr) + \gamma_6 M)$ perpendicular to $pr$. This yields a point $\hat{u}$ that satisfies $rt(p, \hat{u}, r)$ and $d(u, \hat{u}) \leq \gamma_6 M$ by Lemma 2. We do the same for points that have been incorrectly added to $S_2$. Points that were incorrectly discarded are moved to $pr$ (or equivalently $rq$). If $r_1$ is not actually farthest from $pr$, we only have to move it at most $F_{|S_1|}/M$ farther from $pr$ to make it the farthest point. The points $p, r, q$ are not moved as $\widehat{rt}$ correctly classifies them. This gives us $\tilde{S}_1, \tilde{S}_2 \subseteq \tilde{P}$ such that $d_M(\tilde{S}_i, S_i), d_M(\tilde{P}, P) \leq F_{|S_i|}$ and $CH(\tilde{P} \cup \{p, q\}) = \{p\} \cup CH(\tilde{S}_1) \cup \{r\} \cup CH(\tilde{S}_2) \cup \{q\}$.

We now do our induction. The base case is trivial. The induction hypothesis gives us $\hat{H}_i$ such that $d_M(\hat{H}_i, CH(S_i)) \leq 2(D-1)F_{|P|}$. As Hausdorff distance takes maxima, this implies that also

$$d_M(\{p\} \cup \hat{H}_1 \cup \{r\} \cup \hat{H}_2 \cup \{q\}, \{p\} \cup CH(S_1) \cup \{r\} \cup CH(S_2) \cup \{q\}) \leq 2(D-1)F_{|P|} \tag{11.4}$$

By our perturbation analysis $d_M(CH(\tilde{S}_i), CH(S_i)) \leq F_{|S_i|}$, so

$$d_M(\{p\} \cup CH(S_1) \cup \{r\} \cup CH(S_2) \cup \{q\}), \{p\} \cup CH(\tilde{S}_1) \cup \{r\} \cup CH(\tilde{S}_2) \cup \{q\}) \leq \max(F_{|S_1|}, F_{|S_2|}) \leq F_{|P|}. \tag{11.5}$$

By construction of $\tilde{P}, \tilde{S}_1, \tilde{S}_2$, we have

$$CH(\tilde{P} \cup \{p, q\}) = \{p\} \cup CH(\tilde{S}_1) \cup \{r\} \cup CH(\tilde{S}_2) \cup \{q\}, \tag{11.6}$$

and by perturbation analysis

$$d_M(CH(\tilde{P} \cup \{p, q\}), CH(P \cup \{p, q\})) \leq \max(F_{|S_1|}, F_{|S_2|}) \leq F_{|P|}. \tag{11.7}$$

The induction step now follows from combining equations 11.4, 11.5, 11.6, 11.7 using the triangle inequality. □

### Discussion

The worst-case error bound is poor, $O(|P|^2\epsilon)$, if Algorithm 22 is used for the reduction, and if the recursion has depth $O(|P|)$. However, a recursion depth of $O(|P|)$ gives Quickhull a runtime complexity of $O(|P|^2)$, making it useless on that input anyway. If the recursion is reasonably balanced, we have $D \in O(\log(|CH(P)|))$. Under Hypothesis 1 we have also have a far tighter expected bound $\mathbb{E}[F_n] \in O(\epsilon)$. Under these assumptions, which we believe reasonable, the forward error bound will be $O(\log(|CH(P)|)\epsilon)$.

If we do not want to assume Hypothesis 1 holds, we can choose $m \approx \sqrt{n}$ in Algorithm 23 to reduce the bound to $O(\sqrt{|P|}D\epsilon)$. Algorithm 24 further reduces the bound to $O(\log|P|D\epsilon)$, but this significantly complicates the implementation. If the runtime is reasonable, so $D \in O(\log|P|)$, we get the bounds promised in the introduction.

We also note that a parallel implementation of Quickhull will lead to Algorithm 23, for $m$ the number of processors, so parallelisation will improve, rather than worsen, the forward error bound.

## 11.6 Related and Future Work

Finding planar convex hulls is a well-studied problem, and there exist at least ten algorithms for solving it [92, 125, 70, 196, 42, 4, 10, 53, 37, 48].

As far as we are aware, robustness work of the convex hull problem has all been done for variations of Graham Scan. Fortune [76] modifies Graham Scan to

obtain an algorithm with forward error bound of $O(|P|\epsilon)$. Assuming Quickhull has a runtime of $O(|P|\log|P|)$ we obtain slightly worse error bound $O(|P|\log(|P|)\epsilon)$ when using the straightforward reduction algorithm, or a better one when using more complicated reduction algorithms. In practice, we expect an even tighter error bound $O\left(\log(|P|)|\log(|CH(P)|)\epsilon\right)$. For Graham-Fortune the bound will not be much tighter in practice, which we can see as follows. This algorithm first sorts the points by $x$-coordinate and adds the points to a stack in that order. The stack is then popped until the new point makes a right turn with the two points underneath it. The error is linear in the number of successive pops. There are $|P| - |CH(P)|$ points are popped in total, so at least

$$\frac{|P|}{|CH(P)|} - 1$$

points must be popped successively which is large for $|P| \gg |CH(P)|$.

Jaromczyk et al. [124] modify Fortune's algorithm further to a variant that is backward stable with error bound $O(|P|\epsilon)$. Our perturbation analysis shows that this also implies a forward error bound of $O(|P|\epsilon)$. Quickhull does not necessarily have a small backward error because the set we compute may not be convex, and therefore not the solution to any problem. This may be problematic when the hull we compute becomes the input of a different algorithm that expects convex input. In fact, even an output that is truly convex is problematic when we cannot test this at our level of accuracy. To that end Li et al. [159] propose computing strongly convex hulls, where we reject points if their adjacent sides are almost 180°. They provide an algorithm that can construct a strongly convex hull from an approximate one in linear time, which means it can be efficiently combined with Quickhull.

Jiang et al. [127] provide a perturbation analysis for the convex hull problem. They do not scale by $M$ and consider solutions to have an infinite error if they are not a minimal representation. We have chosen to scale because we want to be invariant to units. If a minimal representation is desired, we believe a strongly convex hull should be extracted from Quickhull's output using Li's algorithm. This adds minimal $O(|CH(P)|)$ overhead and reduces the representation further.

The analysis of reduction algorithms is similar to that of summation [108], except in our case some algorithms are uniformly better than others.

## 11.7 Conclusion

We have defined a distance $d_M$ to measure the error between the convex hull and a computed approximation. The perturbation analysis with respect to this distance shows that the convex hull problem is well-conditioned.

We recommend implementing the geometric test $d(u, pq) > d(u', pq)$ as $(q_y - p_y)(u_x - u'_x) < (q_x - p_x)(u_y - u'_y)$ and show that this test is accurate. Error in the Quickhull algorithm as a whole is accumulated through recursion and finding the point $r$ that is farthest from a line segment $pq$. This gives a worst-case bound on the forward error of $O(|P|^2\epsilon)$, where $\epsilon$ is the machine precision. However, we

argue that Quickhull would never be used on data where the recursion is deeper than $O(\log|P|)$, as Fortune-Graham would have superior runtime. Similarly, we do not expect error to accumulate when finding the farthest point $r$ in practice. This gives a more realistic forward error of $O(\log(|CH(P)|)\epsilon)$.

Though we do not believe this is useful in practice, we do provide algorithmic variations that reduce the worst-case bound. By using a different reduction algorithm for finding $r$, we can reduce the error bound to $O(\sqrt{|P|}D\epsilon)$ or $O(D\log(|P|)\epsilon)$, for $D$ the recursion depth. The depth $D$ is at most $|P|$, and $O(\log|P|)$ for data where the runtime complexity is competitive to Graham-Fortune.

## 11.8 Note on Correctness and Shapes

At first glance, it seems strange to put this chapter in a part on rank-polymorphism. The paper this chapter is based on, presents Algorithms 22 to 24 as three different algorithms because that is most suitable for that target audience. However, they were derived as three instances of the same shape-recursive algorithm, in a way that is very similar to the blocked matrix multiplication of Listing 10.4. In fact, so similar that some of the reasoning can be carried over between chapters. The paper [219] that Chapter 10 is based on proves with Agda that blocking does not change the result given associativity of floating point addition (this was stripped out as it is work by a different author and not relevant to the dissertation). Of course, floating point addition is not associative, and the blocking will change the result. In this chapter we explain why we should not care that the two solutions are not bitwise-equal. In terms of numerical stability—the closeness to the Platonic results modelled by floating point numbers—the blocked algorithm is actually more correct [108].

If we would straightforwardly implement Algorithms 22 to 24 in SaC, we would model a point in $\mathbb{R}^2$ as double[2], yielding three functions with signature

```
double[2]
farthest(double[n, 2] P, double[2] p, double[2] q).
```

However, by interpreting P as a higher-dimensional array and letting the shape guide the computation, we can specify one recursive function that implements them all (Listing 11.5). If we call the function on P without reshaping, we get Algorithm 22. If we reshape to double[n / m, m, 2] P, it implements Algorithm 23. And if we raise the rank as high as possible by reshaping to double[2, 2, ..., 2] P, we get the logarithmic Algorithm 24.

If we replace P by {[p] -> A[i, p] * B[p, j]} and the frt (p, q) with +, we have the execution order of computing C[i, j] for rank-recursive matrix multiplication.

I will briefly explain how the argument of Lemma 4 is similar to the error analysis of blocked matrix multiplication, assuming familiarity with [108]. Any summation of elements $S := \{x_1, \cdots, x_n\}$ can be defined as $n-1$ steps of taking two elements of $S$, adding them, and returning the result to $S$. We define $\hat{T}_l$

as the computed sum of step $i$. In the case of matrix multiplication, we have $C_{ij} = \sum_{p=0}^{k} A_{ip} \cdot B_{pj}$, so $x_p = A_{ip} \cdot B_{pj}$. The error bound is $\epsilon \sum_{l=0}^{k} |\hat{T}_l|$. Blocking [k] into [k1, k2] splits a sum of $k_1 k_2$ elements into a sum of $k_1$ subsums of $k_2$ in length, the same way Algorithm 23 splits $P$ into groups of size $m$. The number of additions $x_p$ participates in, is $(k_1 - 1) + (k_2 - 1)$, which is no larger than $(k_1 - 1)(k_2 - 1)$. For this reason, blocking makes the $\hat{T}_l$ smaller, and therefore reduces the a priori (forward) error bound. This is similar to Quickhull, where it reduces the number of comparisons a point participates in.

```
double[2] frt(double[2] p, double[2] q, double[2] u,
              double[2] up)
{
    return ((q[1] - p[1]) * (u[0] - up[0]) <
                (q[0] - p[0]) * (u[1] - up[1])) ?
        u : up;
}

double[2] recursive_farthest(double[2] P,
                             double[2] p, double[2] q)
{
    return P;
}

double[2] recursive_farthest(double[n, d:shp, 2] P,
                             double[2] p, double[2] q)
{
    return with {
        ([0] <= [i] < [n]): recursive_farthest(P[i], p, q);
    }: fold(frt(p, q), p);
}
```

Figure 11.5: Shape Recursive Reduction

# Part III

# Epilogue

# Chapter 12

# Conclusion

This dissertation poses the thesis that *it is time to abstract away hardware architecture in scientific computing.* To that end, we program in languages that do not expose hardware details such as memory and explicit mappings of computations to execution units. Section 12.1 concludes that to a large extent we can generate performant code from such a language for different architectures. We investigate code generation for four types of hardware: CPUs, GPUs, collections of CPUs, and collections of GPUs. Code generation for the latter two are contributions of this work, and are implemented for the language Single-assignment C (SaC).

To design algorithms that are easy to specify and perform well in such a language, we need higher levels of abstractions, notably rank-polymorphism. In Section 12.2, we conclude how rank-polymorphism can help us with controlling locality, numerical error, and with understanding the interplay between multilinearity and parallelism.

## 12.1 Performance Portability

To see to what extent we can portably generate performant code from a minimalist language, we studied a wide range of applications implemented in functional array languages. We can classify applications from the perspective of the hardware, and from the perspective of the algorithm. From the hardware perspective, there are two performance bottlenecks: the speed at which the machine can execute instructions, or the speed at which it can load data. We call these *compute*-bound and *memory*-bound respectively. Implementing an algorithm in a way that is compute-bound, or at least less memory-bound, requires minimizing data movement, which differs between fields. We have looked at dense linear algebra, iterative methods for solving partial differential equations, and signal processing.

A straightforward implementation of an all-pairs N-body simulation is compute-bound due to its high time complexity of $O(N^2)$. This makes it a good representative for the class of compute-bound algorithms. For this benchmark, we have obtained strong evidence in support of the thesis: we can generate code from

a hardware-agnostic SaC implementation that obtains at least 97 % of a hand-optimised baseline on all cores of a CPU, and even 100 % on one core (Chapter 2, Table 2.2). We also obtain near-linear scaling on multiple CPUs (Chapter 6, Fig. 6.1a), and on multiple GPUs (Chapter 7, Fig. 7.7). The code generation for a single GPU does not match the baseline, but other hardware-agnostic languages such as Futhark and DaCe even outperform a handwritten code, without using esoteric language features. Similarly, the SaC baseline does not use language features Futhark and DaCe do not have, suggesting that all these languages can obtain good performance on all these platforms, given enough engineering effort. Therefore, portable performance is possible for this benchmark that we consider representative for compute-bound algorithms.

Matrix-multiplication's time complexity of $O(N\sqrt{N})$ is also high enough to make it compute-bound, but it does require decomposing the computation to work on large blocks at a time. This technique is called *blocking* and is representative for many algorithms in dense-linear algebra. Blocking is implemented differently on CPUs, GPUs, and clusters, but the mathematical idea described above remains the same: a recursive specification that matches the memory hierarchy. It turns out that we can clearly express this using rank-polymorphism in a hardware-independent language, and that this can obtain state-of-the-art performance on a CPU (Chapter 10, Fig. 10.3). We obtain reasonable scaling from the same code on a cluster of CPUs (Chapter 6, Fig. 6.1c). We also obtain good speedups on multiple GPUs (Chapter 7, Fig. 7.4), but the performance on one GPU is not competitive with the state of the art. Table 2.5 shows this is not an inherent problem, as Futhark can get close to the state of the art on GPUs for FlashAttention, which uses matrix-multiplication as its main computational kernel.

Partial differential equations are ubiquitous. For systems that are large enough to necessitate the use of a distributed-memory machine, a time complexity exceeding $O(N \log N)$ is considered too high. This renders the methods memory-bound. A popular method for solving these equations are finite difference or finite element methods [155]. These are solved with functions where each point computes a weighted average of its neighbours. This is called a *stencil*, and is used in MultiGrid (Chapter 2, Table 2.3), and Full MultiGrid (Chapter 6, Fig. 6.1b). We see good results for MultiGrid on CPUs and GPUs, but run into low-level difficulties for distributed systems, both of CPUs and GPUs (Chapter 7, Fig. 7.9). Namely, we obtain less bandwidth for memory that is accessible by multiple processors. For differential equations with a less regular grid, these linear functions are represented as sparse matrices. This is a difficult case for our approach, as the Conjugate Gradient benchmark shows (Chapter 5), and future work is needed. Chapter 13 will discuss this in detail.

Methods to construct irregular grids are usually combinatorial. Combinatorial methods focus on constructing data structures and rearranging values, rather than computing new values. As an example, we take Quickhull, an algorithm for computing the convex hull. Table 2.4 shows this is a challenging benchmark for the compilers of hardware-agnostic array languages we have now. In Chapter 13 we discuss some language constructs that may be a better fit than array languages in this area.

## 12.2 Rank-Polymorphic Algorithm Design

To make optimal use of a language that abstracts away hardware architecture, we should use higher-level constructs to control parallelism and data movement. This is important for portable performance, and also makes code easier to understand. It turns out rank-polymorphism—available in languages such as SaC—is a particularly useful feature for these kinds of programming patterns.

Rank polymorphism is useful for designing algorithms for the Fourier transform and related functions. Among many other applications, these functions can be used as spectral methods for solving partial differential equations. Spectral methods are similar to the finite difference methods discussed previously, but whereas finite difference methods and corresponding stencil functions take a local view, spectral methods and the Fourier transform take a global view. In other words, we look at how each cell affects each other cell, rather than only its neighbours. This mix of long range and short range effects makes it interesting and challenging to expose parallelism while minimizing data movement in the Fourier transform. In higher dimensions, this becomes even more complicated. Chapter 8 shows rank-polymorphism is a powerful tool for understanding how higher dimensions and parallelism interact. The rank-polymorphic algorithm that this chapter contributes, can be implemented efficiently in C (Chapter 9). An implementation in a language with support for rank-polymorphism is left for future work.

In the Fourier transform, rank-polymorphism arises from multilinearity, and is used to derive algorithms in a rank-independent way. We can also use rank-polymorphism to implement an efficient matrix-multiplication, but here we use it to model recursion. The basic idea behind optimising matrix-multiplication on CPUs, GPUs, and distributed systems, is to compute matrix-multiplication recursively, so that the input and output of recursive calls fit in a level on the memory hierarchy. However, implementations of this idea vary drastically between architectures, and the specific recursion (the optimal structure differs between architectures) is hard coded in the function implementation. Chapter 10 shows that rank-polymorphism can capture the entire class of recursive algorithms, and that it is possible to generate multithreaded code with no performance penalty.

Recursive matrix-multiplication changes the order in which we perform floating point operations, which changes the computed result. We see that rank-polymorphism can help us control these numerical errors as well. We do so for Quickhull (Chapter 11), and Section 11.8 shows that a core computation in the Quickhull algorithm is a minor variation on rank-polymorphic matrix-multiplication. By the same argument, we show that the rank-polymorphic matrix-multiplication is more accurate.

## 12.3 Implications

Programming in a hardware-agnostic language lets us take advantage of parallel processors without needing to understand how they work. This decreases the runtime of programs, but does not necessarily increase the maximum problem size

we can solve. For that, we would need a collection of processors that can pool their memory together to collaboratively compute a task, and code that makes effective usage of the combined memory. With the contributions of this work, it is now possible to generate such code for collections of processors from a hardware-agnostic language as well.

To take advantage of these hardware-agnostic languages, we should learn how we can leverage its features to optimise the overall program structure in a hardware-agnostic way as well. We show that rank-polymorphism is a surprisingly versatile tool for this. We can use it to control data movement in a hardware-independent way for recursive algorithms, but also to understand how multilinearity interacts with parallelism, or to control numerical error.

# Chapter 13

# Future Work

The goal of this dissertation was to abstract over parallel architectures the way FORTRAN abstracted over instruction sets. We took a minimal approach that focusses on array languages. A key challenge in abstracting over architectures, is that the best algorithm may depend on the target hardware, and that this may require significant domain knowledge. We identified rank-polymorphism as a powerful tool for tackling this challenge, but it is only applicable to certain classes of problems. In this chapter, we identify some opportunities for future work in other problem classes.

## 13.1 C-Compiler Improvements

Abstracting over parallel architectures makes the compiler responsible for lower-level optimisations. Chapter 10 shows that the step from generated C to assembly is not yet good enough to compete with BLAS, so we had to use the foreign function interface. The reason for this, is that the C compilers at the time did not use the registers as cache effectively. We had to write Listing 10.7 in C with GCC compiler extensions to get the assembly we needed.

However, compilers of lower-level languages continuously become better at auto-vectorisation and register allocation. This is already visible in the few years since writing the paper [219] Chapter 10 is based on: Clang 16.0.0 (with flags `-O3 -mavx2 -mfma`), generates the same code from Listing 13.1 as from Listing 10.7. This code is pure C, and similar to what SaC generates, which suggests that it may soon be unnecessary to use the foreign function interface as workaround.

So we can rely on C-compiler writers to handle the future work that is needed for generating SIMD instructions and register tiling.

## 13.2 Algorithmic Variants

Chapters 3 and 4 show that we can obtain state-of-the-art performance on both CPUs and GPUs with the same algorithmic idea. However, this is not always

```
void matmul(double **cp, double *a, double *b,
            int offset_a, int offset_b)
{
  double *c = calloc(MR * NR, sizeof(double));
  a += offset_a; b += offset_b;
  for (int k = 0; k < KC; k++) {
    for (int i = 0; i < MR; i++) {
      for (int j = 0; j < NR; j++) {
        c[i * NR + j] += a[i * KC + k] * b[k * NR + j];
      }
    }
  }
  *cp = c;
}
```

Listing 13.1: Handwritten kernel

possible. For example, on a purely sequential processor (that you would be hard-pressed to find), Hoare's algorithm for partitioning an array is simpler and likely faster. Rank-polymorphism is a tool for expressing algorithmic variations, but it is no silver bullet.

As a practical example where future work is needed to deal with different algorithms, let us look at the number of iterations needed for the conjugate gradient method to converge. This is proportional to $\sqrt{\kappa(A)}$, where $Ax = b$ is the linear system we try to solve, and $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ is the condition number of $A$. As the condition number can get very large (for the system MG solves, the condition number is quadratic in $h$), solving $Ax = b$ directly is impractical. For this reason, we typically solve the equivalent system $(M^{-1}A)y = M^{-1}b$ such that $M^{-1}A$ has a better condition number. We want $M^{-1}$ to be easy to compute, and approximately invert $A$. This is called *preconditioning*. The library PETSc uses incomplete Cholesky decomposition as a preconditioner for sequential execution by default. For parallel execution, incomplete Cholesky decomposition is not even implemented because it cannot be effectively parallelised. Instead, parallel execution uses the blocked Jacobi algorithm by default, which is an $M^{-1}$ of the form

$$\begin{pmatrix} M_0^{-1} & & & \\ & M_1^{-1} & & \\ & & \ddots & \\ & & & M_{p-1}^{-1} \end{pmatrix}$$

on $p$ processors. This ensures the preconditioner needs no communication, which outweighs blocked Jacobi being less effective in reducing the condition number.

Any of these algorithms can be specified in a hardware-agnostic way, and the generated code may be good for that algorithm, but the *choice* of algorithm cannot be done in a hardware-agnostic way. It is an open question how to best deal

with this. As a possible approach, a language could allow the programmer to supply multiple algorithms for the same goal (e.g. multiple preconditioners). The compiler could then choose which to run with heuristics, profiling, or a mix of the two.

## 13.3 Domain Specific Languages

The goal of this dissertation is to abstract over parallel architectures, but the approach abstracts over something else too: the problem domain. General purpose languages such as SaC can be used for many fields in scientific computing. A language that does not make this abstraction, but is specifically designed for a problem domain is called a *domain-specific language* or DSL. In such a language, it is possible to hard code domain-specific knowledge into the compiler, for example the condition number of the previous section. This may lead to more efficient code, but also comes at a cost: a new language has to be designed, a compiler has to be written, and projects using this DSL become more complicated because they mix languages.

### Sparse Linear Algebra

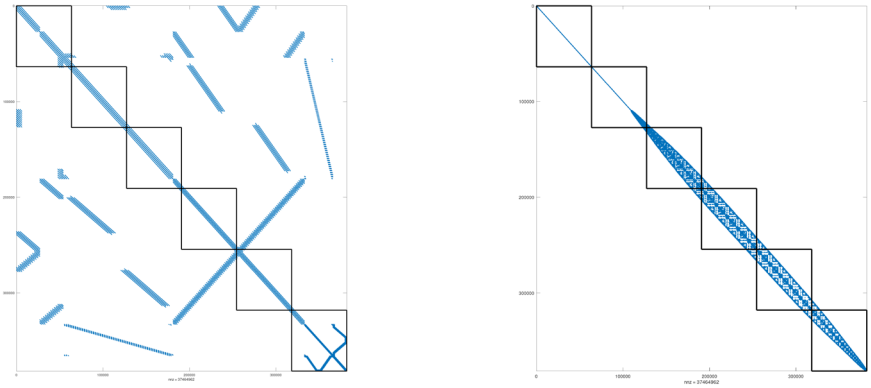
Sparse linear algebra is an interesting area for DSLs. Let us look at an example that illustrates why we cannot shift algorithmic improvements to the compiler of a generic language.

The conjugate gradient method from Section 5.5 is iterative and relies on sparse matrix-vector multiplication (spmv) as main computational kernel. To understand the poor performance of the conjugate gradient method in Chapter 5, we have to look at data movement. Figure 13.1 is a sparse matrix corresponding to a partial differential equation that describes how fluid behaves in a jet engine compressor [62]. The conjugate gradient method repeatedly multiplies this sparse matrix $A$ with a dense vector. Each non-zero $A[i, j]$ corresponds to a read from position $j$ of the input vector and a write to position $i$ of the output vector. This means accesses are local if both $i$ and $j$ are between `ShrayStart(x)` and `ShrayEnd(x)`. These are the non-zeroes in the squares of Figure 13.1a. We see that many accesses are non-local, and this explains the limited speedup we observed in Subsection 5.5.

As a possible approach, we can observe that reordering the basis elements does not lead to a different result, so we can attempt to bring the non-zeroes closer to the diagonal. The (reverse) Cuthill-McKee algorithm [58] is effective as you can see in Figure 13.1b, and can be implemented in distributed memory [14]. This does not only reduce data movement between nodes, but also between RAM and caches as there is more reuse of the vector. Similarly to how the recursive matrix multiplication algorithm of Chapter 10 is also used for distributed computers [84], this optimisation is useful for anything with a memory hierarchy. This makes the overhead of finding the basis reordering well worth it in this case.

However, the effectiveness depends on the sparsity structure of the matrix: the bandwidth (maximal distance of non-zeroes to the diagonal) cannot always be

meaningfully reduced, for example in the case of random matrices. In that case, it would be a waste of time and energy to reorder the basis elements.



(a) Default order

(b) Cuthill-McKee permutation

Figure 13.1: Bandwidth reduction of sparse matrix RM07R with the reverse Cuthill-McKee permutation leads to less communication in a distributed setting.

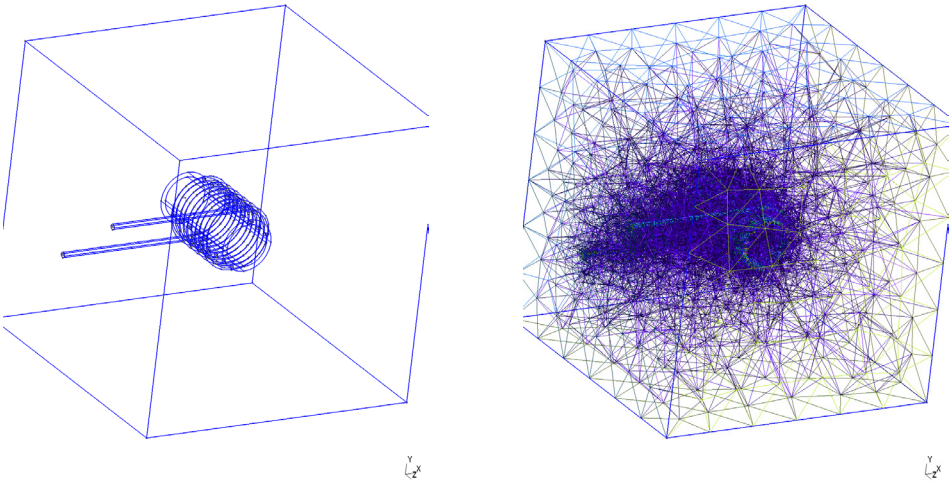
In general it is not possible to statically decide the best choice, but there are application areas, such as finite-element discretisations, where we can generate matrices of different sizes with a similar sparsity structure. An opportunity for future work would be to design a DSL for sparse linear algebra that can take a small matrix as argument at compile time. The sparsity structure of the small matrix can be analysed quickly, and be used by the compiler to generate fast code for larger matrices of similar structure.

## Computational Geometry

The MultiGrid method of Chapter 2 solves $\nabla^2 u = v$ for $v$ some function on a cubical domain $[0,1]^3$. The discretisation of $v$ takes a regular grid of samples, so an $n \times n \times n$ array $V$ defined by $V[i,j,k] = v(i \cdot h, j \cdot h, k \cdot h), \quad h = \frac{1}{n}$. A discrete version of $\nabla^2$ is a linear function $\mathbb{R}^{n \times n \times n} \to \mathbb{R}^{n \times n \times n}$ and can be described by a matrix $A$ that has seven non-zero diagonals: one with $-6/h^2$, and six with $1/h^2$. MultiGrid then solves the equation $AU = V$ for $U$. We do not form $A$ explicitly as this is just a stencil operation. This system is very suitable for array languages and automatic parallelisation.

To solve the PDE to the same level of accuracy with less memory and computation, we could take more samples in a region where $v$ varies rapidly. We may also have a function $v$ that is defined piecewise on a more complicated domain modelling a physical object as in Figure 13.2. In both cases, the resulting discretisation is not a regular array, but a simplicial complex. These are best described by data structures such as simplex trees [35] that are hard to implement efficiently

in an array language. The algorithms for computing them are combinatorial in nature, which are harder to parallelise.



(a) Model (b) Grid

Figure 13.2: Model of a metal coil in a tube, and a grid that is used to solve the partial differential equation describing heat transfer. Generated with gmsh [85].

One of these algorithms is Quickhull, and Chapters 2 and 4 show that it is not (yet) possible for functional array languages to obtain good performance for here. However, by using the same idea on GPUs (Chapter 3) with theory from distributed computing, we show that the main idea is not hardware-specific. For computational geometry, especially in higher dimensions, there is no straightforward way to represent important data structures as arrays, at least not arrays whose index sets are Cartesian products. Future work can investigate array languages that support irregular arrays, but one could also design a DSL where simplicial complexes are first class citizens. It is an open question whether we need data parallelism, task parallelism, parallelised higher-order functions, or a combination in this area.

## 13.4 Rank-Polymorphic Algorithm Design

Domain specific languages are widely used for the FFT and related transform processing [81, 176]. However, Chapters 8 and 10 suggest that both the recursion of the one-dimensional FFT and multilinearity of the higher-dimensional FFT can be captured using rank-polymorphism. It would be interesting to see if we really need a DSL here, or if rank-polymorphism can express a performance-portable specification in a hardware-agnostic language, similar to matrix-multiplication.

Chapter 10 shows we can match state-of-the-art matrix-multiplication with rank-polymorphism, but matrix-multiplication is far from the only blocked algorithm. Almost all routines in (Sca)LAPACK [29] are blocked algorithms, so it would be interesting to see if rank-polymorphism can offer benefits there also. Some of these algorithms are more challenging because the blocking changes between phases of the algorithm.

## 13.5 Non-Academic Future Work

This dissertation shows hardware-agnostic languages can obtain good performance on a range of benchmarks. We demonstrate advantages of programming at this higher level of abstraction. Chapter 2 shows that at least one functional array language can obtain good performance across a wide range of benchmarks, without using esoteric language features that would prevent a different language from implementing the same optimisations. It also shows that future work is needed to ensure all languages obtain good performance. However, these software engineering tasks, together with developing an ecosystem, contain little novelty. This makes it future work for actors outside of academia, and this dissertation makes an argument that it would be useful for them to do so.

# Bibliography


[1] J. Aaldering, S.-B. Scholz, and B. van Gastel. "Type Patterns: Pattern Matching on Shape-Carrying Array Types". In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. 2024. DOI: 10.1145/3652561.3652572.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. ia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X Zhen. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[3] P. Achten and R. Plasmeijer. "The ins and outs of Clean I/O". In: *Journal of Functional Programming* 5 (1995). DOI: 10.1017/S0956796800001258.

[4] S.G. Akl and G.T. Toussaint. "A Fast Convex Hull Algorithm". In: *Information Processing Letters* 7 (1978). DOI: 10.1016/0020-0190(78)90003-0.

[5] J.R. Allen and K. Kennedy. "Automatic Loop Interchange". In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. 1984. DOI: 10.1145/502874.502897.

[6] M.S. Alnæs, A. Logg, K.B. Ølgaard, M.E. Rognes, and G.N. Wells. "Unified form language: A domain-specific language for weak formulations of partial differential equations". In: *ACM Trans. Math. Softw.* 40.2 (2014). DOI: 10.1145/2566630.

[7] D. Anderson, G.E. Blelloch, L. Dhulipala, M. Dobson, and Y. Sun. "The problem-based benchmark suite (PBBS), V2". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022. DOI: 10.1145/3503221.3508422.

[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Third. 1999. ISBN: 0-89871-447-8 (paperback).

[9] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C.E. Oancea. "FinPar: A Parallel Financial Benchmark". In: *ACM Trans. Archit. Code Optim.* 13 (2016). DOI: 10.1145/2898354.

[10] A.M. Andrew. "Another efficient algorithm for convex hulls in two dimensions". In: *Information Processing Letters* 9 (1979). DOI: 10.1016/0020-0190(79)90072-3.

[11] G. Araujo, D. Griebler, D.A. Rockenbach, M. Danelutto, and L.G. Fernandes. "NAS Parallel Benchmarks with CUDA and beyond". In: *Software: Practice and Experience* 53 (2023). DOI: https://doi.org/10.1002/spe.3056.

[12] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. "In-Place Parallel Super Scalar Samplesort (IPSSSSo)". In: *25th Annual European Symposium on Algorithms (ESA 2017)*. 2017. DOI: 10.4230/LIPIcs.ESA.2017.9.

[13] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra. "heFFTe: Highly Efficient FFT for Exascale". In: *Computational Science – ICCS 2020*. 2020. DOI: 10.1007/978-3-030-50371-0_19.

[14] A. Azad, M. Jacquelin, A. Buluç, and E.G. Ng. "The Reverse Cuthill-McKee Algorithm in Distributed-Memory". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017. DOI: 10.1109/IPDPS.2017.85.

[15] J. Backus. "The history of Fortran I, II, and III". In: *History of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1978. Chap. 2. ISBN: 0127450408. DOI: 10.1145/800025.1198345.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. "The NAS parallel benchmarks—summary and preliminary results". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 1991. DOI: 10.1145/125826.125925.

[17] S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith. "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries". In: *Modern Software Tools in Scientific Computing*. 1997. DOI: 10.1007/978-1-4612-1986-6_8.

[18] D. van Balen, T. De Matteis, C. Grelck, T. Henriksen, A.W. Hsu, G.K. Keller, T.J. Koopman, T.L. McDonell, C. Oancea, S.-B. Scholz, A. Sinkarovs, T. Smeding, P. Trinder, I.G. de Wolff, and A.N. Ziogas. *Comparing Parallel Functional Array Languages: Programming and Performance*. 2025. arXiv: 2505.08906 [cs.PL]. URL: https://arxiv.org/abs/2505.08906.

[19] J. Barkley Rosser. "Nine-point difference solutions for Poisson's equation". In: *Computers and Mathematics with Applications* 1 (1975). DOI: 10.1016/0898-1221(75)90035-8.

[20] J. Barnes and P. Hut. "A hierarchical O (N log N) force-calculation algorithm". In: *nature* 324 (1986).

[21] T. Ben-Nun, J. de Fine Licht, A.N. Ziogas, T. Schneider, and T. Hoefler. "Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019. DOI: 10.1145/3295500.3356173.

[22] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver Elbert, Rhea George, Jeremy McGibbon, Lukas Trümper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. "Productive Performance Engineering for Weather and Climate Modeling with Python". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. 2022. ISBN: 9784665454445. DOI: 10.1109/SC41404.2022.00078.

[23] L. Bergstrom and J. Reppy. "Nested Data-parallelism on the Gpu". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. Copenhagen, Denmark, 2012. DOI: 10.1145/2364527.2364563.

[24] P. van Beurden, T.J. Koopman, and S.-B. Scholz. *Artifact for the paper: Multi-GPU Code Generation for Out-Of-Core Problems*. 2025. DOI: 10.5281/zenodo.17036308.

[25] P. van Beurden and S.-B. Scholz. "On Generating Out-Of-Core GPU Code for Multi-Dimensional Array Operations". In: *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*. 2023. DOI: 10.1145/3587216.3587223.

[26] P. van Beurden, T.J. Koopman, and S.-B. Scholz. "Multi-GPU Code Generation for Out-of-Core Problems". In: *Trends in Functional Programming*. 2025. DOI: 10.1007/978-3-031-99751-8_6.

[27] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. UK: Oxford University Press, 2020. ISBN: 9780198788348. DOI: 10.1093/oso/9780198788348.001.0001.

[28] M. Blacher, J. Giesen, Sanders. P., and J. Wassenberg. *Vectorized and performance-portable Quicksort*. 2022. DOI: 10.1002/spe.3142.

[29] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-397-8 (paperback).

[30] L.S. Blackford, A. Petitet, R. Pozo, K. Remington, R.C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28 (2002). DOI: 10.1145/567806.567807.

[31] G.E. Blelloch. "Programming parallel algorithms". In: *Commun. ACM* 39 (1996). DOI: 10.1145/227234.227246.

[32] G.E. Blelloch. "Scans as primitive parallel operations". In: *IEEE Transactions on Computers* 38.11 (1989). DOI: 10.1109/12.42122.

[33] Guy E Blelloch. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge, 1990.

[34] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer". In: *SIGARCH Comput. Archit. News* 22 (1994). DOI: 10.1145/192007.192024.

[35] J.-D. Boissonnat and C. Maria. "The Simplex Tree: An Efficient Data Structure for General Simplicial Complexes". In: *Algorithmica* 70 (2014). DOI: 10.1007/s00453-014-9887-3.

[36] S. Borowski and T. Klüner. "Massively parallel Hamiltonian action in pseudospectral algorithms applied to quantum dynamics of laser induced desorption". In: *Chemical Physics* 304 (2004). DOI: 10.1016/j.chemphys.2004.06.012.

[37] B.C. Bradford, D.P. Dobkin, and H. Huhdanpaa. "The quickhull algorithm for convex hulls". In: *ACM Transactions on Mathematical Software* 22 (1996). DOI: 10.1145/235815.235821.

[38] B. Bramas. "A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake". In: *International Journal of Advanced Computer Science and Applications* 8 (2017). DOI: 10.14569/IJACSA.2017.081044.

[39] A. Brandt. "Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems". In: *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*. 1973. DOI: 10.1007/BFb0118663.

[40] J.S. Bridle. "Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition". In: *Neurocomputing*. Ed. by Françoise Fogelman Soulié and Jeanny Hérault. 1990. DOI: 10.1007/978-3-642-76153-9_28.

[41] A. Buluç, J.T. Fineman, M. Frigo, J.R. Gilbert, and C. E. Leiserson. "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks". In: *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*. 2009. DOI: 10.1145/1583991.1584053.

[42] A. Bykat. "Convex hull of a finite set of points in two dimensions". In: *Information Processing Letters* 7 (1978). DOI: 10.1016/0020-0190(78)90021-2.

[43] D. Callahan, B.L. Chamberlain, and H.P. Zima. "The cascade high productivity language". In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* 2004. DOI: 10.1109/HIPS.2004.1299190.

[44] D. Cederman and P. Tsigas. "GPU-Quicksort: A practical Quicksort algorithm for graphics processors". In: *ACM J. Exp. Algorithmics* 14 (2010). DOI: 10.1145/1498698.1564500.

[45] M.T. Chakravarty, G. Keller, S. Lee, T.L. McDonell, and V. Grover. "Accelerating Haskell array codes with multicore GPUs". In: *DAMP: Declarative Aspects of Multicore Programming*. 2011. DOI: 10.1145/1926354.1926358.

[46] M.T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. "Associated type synonyms". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. 2005. DOI: 10.1145/1086365.1086397.

[47] M.T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. "Data parallel Haskell: a status report". In: *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*. 2007. DOI: 10.1145/1248648.1248652.

[48] T.M. Chan. "Optimal output-sensitive convex hull algorithms in two and three dimensions". In: *Discrete & Computational Geometry* 16 (1996). DOI: 10.1007/BF02712873.

[49] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: an object-oriented approach to non-uniform cluster computing". In: *SIGPLAN Not.* 40.10 (2005). DOI: 10.1145/1103845.1094852.

[50] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh. "Maliciously secure matrix multiplication with applications to private deep learning". In: *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*. 2020. DOI: 10.1007/978-3-030-64840-4_2.

[51] Wei-Yu Chen, C. Iancu, and K. Yelick. "Communication optimizations for fine-grained UPC applications". In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 2005. DOI: 10.1109/PACT.2005.13.

[52] S. Chien, I. Peng, and S. Markidis. "Performance Evaluation of Advanced Features in CUDA Unified Memory". In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 2019. DOI: 10.1109/MCHPC49590.2019.00014.

[53] K.L. Clarkson, K. Mehlhorn, and R. Seidel. "Four results on randomized incremental constructions". In: *Computational Geometry* 3 (1993). DOI: 10.1016/0925-7721(93)90009-U.

[54] Morten Tychsen Clausen. "Regular Segmented Single-pass Scan in Futhark". MA thesis. Department of Computer Science, Faculty of Science, University of Copenhagen, https://futhark-lang.org/student-projects/morten-msc-thesis.pdf, 2021. URL: https://futhark-lang.org/student-projects/morten-msc-thesis.pdf.

[55] M.B. Cohen, Y.T. Lee, and Z. Song. "Solving Linear Programs in the Current Matrix Multiplication Time". In: *J. ACM* 68 (2021). DOI: 10.1145/3424305.

[56] J.W. Cooley and J.W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19 (1965).

[57] C. Cortes and V. Vapnik. "Support-vector networks". In: *Machine learning* 20 (1995). DOI: 10.1007/BF00994018.

[58] E. Cuthill and J. McKee. "Reducing the bandwidth of sparse symmetric matrices". In: *Proceedings of the 1969 24th National Conference*. 1969. ISBN: 9781450374934. DOI: 10.1145/800195.805928.

[59] L. Dalcin, M. Mortensen, and D.E. Keyes. "Fast parallel multidimensional FFT using advanced MPI". In: *Journal of Parallel and Distributed Computing* 128 (2019). DOI: 10.1016/j.jpdc.2019.02.006.

[60] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems*. Vol. 35. 2022. DOI: doi/10.5555/3600270.3601459.

[61] Dao AI Lab. *FlashAttention*. https://github.com/Dao-AILab/flash-attention. 2024.

[62] T.A. Davis and Y. Hu. "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38 (2011). DOI: 10.1145/2049662.2049663.

[63] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. "Partitioned Global Address Space Languages". In: *ACM Comput. Surv.* 47 (2015). DOI: 10.1145/2716320.

[64] G.S. Delp, D.J. Farber, R.G. Minnich, J.M. Smith, and M.-C. Tam. "Memory as a network abstraction". In: *IEEE Network* 5 (1991). DOI: 10.1109/65.93183.

[65] Daniel Di Domenico, Gerson G. H. Cavalheiro, and João V. F. Lima. "NAS Parallel Benchmark Kernels with Python: A performance and programming effort analysis focusing on GPUs". In: *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2022. DOI: 10.1109/PDP55904.2022.00013.

[66] C.H.Q. Ding, R.D. Ferraro, and D.B. Gennery. "A Portable 3D FFT Package for Distributed-Memory Parallel Architectures". In: *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computina*. 1995.

[67] M. Diogo and C. Grelck. "Towards Heterogeneous Computing without Heterogeneous Programming". In: *Trends in Functional Programming*. 2013. DOI: 10.1007/978-3-642-40447-4_18.

[68] J. Dongarra and F. Sullivan. "Guest Editors Introduction to the top 10 algorithms". In: *Computing in Science and Engineering* 2.1 (2000). DOI: 10.1109/MCISE.2000.814652.

[69] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. "Cashmere-VLM: Remote memory paging for software distributed shared memory". In: *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*. 1999. DOI: 10.1109/IPPS.1999.760451.

[70] W.F. Eddy. "A New Convex Hull Algorithm for Planar Sets". In: *ACM Trans. Math. Softw.* 3 (1977). DOI: 10.1145/355759.355766.

[71] S. Edelkamp and A. Weiß. "BlockQuicksort: Avoiding Branch Mispredictions in Quicksort". In: *ACM J. Exp. Algorithmics* 24 (2019). DOI: 10.1145/3274660.

[72] M. El Kharroubi, B. Coudray, and O. Malaspinas. "Distributed parallel computing with Futhark: a functional language to generate distributed parallel code". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2022. DOI: 10.1145/3520306.3534501.

[73] E. Feig and S. Winograd. "On the multiplicative complexity of discrete cosine transforms". In: *IEEE Transactions on Information Theory* 38.4 (1992). DOI: 10.1109/18.144722.

[74] J.T. Feo, D.C. Cann, and R.R. Oldehoeft. "A report on the sisal language project". In: *Journal of Parallel and Distributed Computing* 10.4 (1990). DOI: 10.1016/0743-7315(90)90035-N.

[75] B. Fleisch and G. Popek. "Mirage: A Coherent Distributed Shared Memory Design". In: *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*. 1989. DOI: 10.1145/74850.74871.

[76] S. Fortune. "Stable maintenance of point set triangulations in two dimensions". In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. 1989. DOI: 10.1109/SFCS.1989.63524.

[77] Message Passing Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Available at https://www.mpi-forum.org/docs/. USA, 1994.

[78] Ian T. Foster and Patrick H. Worley. "Parallel Algorithms for the Spectral Transform Method". In: *SIAM Journal on Scientific Computing* 18 (1997). DOI: 10.1137/S1064827594266891.

[79] R.S Francis and L.J.H Pannan. "A parallel partition for enhanced parallel QuickSort". In: *Parallel Computing* 18 (1992). DOI: 10.1016/0167-8191(92)90089-P.

[80] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G.S. Blair, and A. Friday. "The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations". In: *Patterns* 2 (2021). DOI: 10.1016/j.patter.2021.100340.

[81] M. Frigo and S.G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93 (2005). DOI: 10.1109/JPROC.2004.840301.

[82] Matteo Frigo. “A fast Fourier transform compiler”. In: *SIGPLAN Not.* 34 (1999). DOI: 10.1145/301631.301661.

[83] R. Gareev, T. Grosser, and M. Kruse. “High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach”. In: *ACM Transactions on Architectural Code Optimization* 15 (2018). DOI: 10.1145/3235029.

[84] R.A. van de Geijn and J. Watts. “SUMMA: Scalable universal matrix multiplication algorithm”. In: *Concurrency: Practice and Experience* 9 (1997).

[85] Christophe Geuzaine and Jean-François Remacle. “Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities”. In: *International journal for numerical methods in engineering* 79.11 (2009).

[86] T.A. El-Ghazawi and L. Smith. “UPC: unified parallel C”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (2006). DOI: doi/10.1145/1188455.1188483.

[87] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. “Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies”. In: *International Journal of Parallel Programming* 34 (2006). DOI: 10.1007/s10766-006-0012-3.

[88] J. Glaser, T.D. Nguyen, J.A. Anderson, P. Lui, F. Spiga, J.A. Millan, D.C. Morse, and S.C. Glotzer. “Strong scaling of general-purpose molecular dynamics simulations on GPUs”. In: *Computer Physics Communications* 192 (2015). DOI: doi.org/10.1016/j.cpc.2015.02.028.

[89] GNU Project. *Using Vector Instructions through Built-in Functions*. https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html, Last accessed on 2023-05-30.

[90] M. González and E. Morancho. “Multi-GPU systems and Unified Virtual Memory for scientific applications: The case of the NAS multi-zone parallel benchmarks”. In: *Journal of Parallel and Distributed Computing* 158 (2021). DOI: 10.1016/j.jpdc.2021.08.001.

[91] K. Goto and R.A. van de Geijn. “Anatomy of High-Performance Matrix Multiplication”. In: *ACM Transactions on Mathematical Software* 34 (2008). DOI: 10.1145/1356052.1356053.

[92] R.L. Graham. “An efficient algorithm for determining the convex hull of a finite planar set”. In: *Information Processing Letters* 1 (1972). DOI: 10.1016/0020-0190(72)90045-2.

[93] C. Grelck. “Single Assignment C (SAC): The Compilation Technology Perspective”. In: *6th Central European Functional Programming Summer School (CEFP’15), Budapest, Hungary*. 2019. DOI: 10.1007/978-3-030-28346-9_7.

[94] C. Grelck and S.-B. Scholz. “Classes and Objects As Basis for I/o in Sac”. In: *7th International Workshop on Implementation of Functional Languages (IFL’95), Båstad, Sweden*. 1995.

[95] C. Grelck and S.-B. Scholz. "Merging Compositions of Array Skeletons in SAC". In: *Journal of Parallel Computing* 32 (2006). DOI: 10.1016/j.parco.2006.08.003.

[96] C. Grelck and S.B. Scholz. "SAC - A Functional Array Language for Efficient Multi-threaded Execution". In: *Int. J. Parallel Program.* 34.4 (2006). DOI: 10.1007/S10766-006-0018-X.

[97] C. Grelck and K. Trojahner. "Implicit Memory Management for SaC". In: *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*. Ed. by C. Grelck and F. Huch. 2004.

[98] Y. Gu, O. Obeya, and J. Shun. "Parallel In-Place Algorithms: Theory and Practice". In: *Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 2021. DOI: 10.1137/1.9781611976489.9.

[99] J. Guo, J. Thiyagalingam, and S.-B. Scholz. "Breaking the GPU programming barrier with the auto-parallelising SAC compiler". In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. 2011. DOI: 10.1145/1926354.1926359.

[100] D.A. Ham, P. H. J. Kelly, L. Mitchell, C. J. Cotter, R. C. Kirby, K. Sagiyama, N. Bouziani, S. Vorderwuelbecke, T. J. Gregory, J. Betteridge, D. R. Shapero, R. W. Nixon-Hill, C. J. Ward, P. E. Farrell, P. D. Brubeck, I. Marsden, T. H. Gibson, M. Homolya, T. Sun, A. T. T. McRae, F. Luporini, A. Gregory, M. Lange, Simon W. Funke, F. Rathgeber, G.-Teodor Bercea, and G. R. Markall. *Firedrake User Manual*. First edition. Imperial College London et al. May 2023. DOI: 10.25561/104839.

[101] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. Oancea. "Compiling Generalized Histograms for GPU". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020. DOI: 10.1109/SC41405.2020.00101.

[102] T. Henriksen, K.F. Larsen, and C.E. Oancea. "Design and GPGPU Performance of Futhark's Redomap Construct". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 2016. DOI: 10.1145/2935323.2935326.

[103] T. Henriksen and C.E. Oancea. "A T2 Graph-reduction Approach to Fusion". In: *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*. 2013. DOI: 10.1145/2502323.2502328.

[104] T. Henriksen, N.G.W. Serup, M. Elsman, F. Henglein, and C.E. Oancea. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017. DOI: 10.1145/3062341.3062354.

[105] T. Henriksen, F. Thorøe, M. Elsman, and C.E. Oancea. "Incremental Flattening for Nested Data Parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 2019. DOI: 10.1145/3293883.3295707.

[106] M.R. Hestenes, E. Stiefel, et al. "Methods of conjugate gradients for solving linear systems". In: *Journal of research of the National Bureau of Standards* 49.6 (1952).

[107] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Society for Industrial and Applied Mathematics, 2002. DOI: 10.1137/1.9780898718027.

[108] Nicholas J. Higham. "The Accuracy of Floating Point Summation". In: *SIAM Journal on Scientific Computing* 14.4 (1993). DOI: 10.1137/0914050.

[109] N.-D. Hoang and N.K. Linh. "Quicker than quickhull". In: *Vietnam Journal of Mathematics* 43 (2015).

[110] C.A.R. Hoare. "Quicksort". In: *The computer journal* 5 (1962).

[111] J. Hoberock and N. Bell. *Thrust: A Parallel Template Library*. 2010. URL: http://thrust.github.io.

[112] T. Hoefler, A. Lumsdaine, and W. Rehm. "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI". In: *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. 2007. DOI: 10.1145/1362622.1362692.

[113] S. Holst Larsen. *Multi-GPU Futhark Using Parallel Streams*. 2019. URL: https://futhark-lang.org/student-projects/steffen-msc-thesis.pdf.

[114] A.W. Hsu. "A data parallel compiler hosted on the GPU". PhD thesis. Indiana University, 2019. URL: https://hdl.handle.net/2022/24749.

[115] A.W. Hsu. "Accelerating information experts through compiler design". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. 2015. DOI: 10.1145/2774959.2774968.

[116] J. Huang, T.M. Smith, G.M. Henry, and R.A. Van De Geijn. "Strassen's Algorithm Reloaded". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016. DOI: 10.1109/SC.2016.58.

[117] R.K.W. Hui and M.J. Kromberg. "APL since 1978". In: *Proc. ACM Program. Lang.* 4 (2020). DOI: 10.1145/3386319.

[118] R. Ihaka and R. Gentleman. "R: A Language for Data Analysis and Graphics". In: *Journal of Computational and Graphical Statistics* 5 (1996). DOI: 10.1080/10618600.1996.10474713.

[119] M.A. Inda. "Construction Parallel Algorithms for Discrete Transforms". PhD thesis. Utrecht University, 2000.

[120] M.A. Inda and R.H. Bisseling. "A simple and efficient parallel FFT algorithm using the BSP model". In: *Parallel Computing* 27.14 (2001).

[121] Intel. *Intel oneAPI Math Kernel Library*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, Last accessed on 2023-05-31.

[122] K.E Iverson. "A programming language". In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. 1962.

[123] N. Janssen and S.-B. Scholz. "On Mapping N-Dimensional Data-Parallelism Efficiently into GPU-Thread-Spaces". In: *Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages*. 2022. DOI: 10.1145/3544885.3544894.

[124] J.W. Jaromczyk and G.W. Wasilkowski. "Computing convex hull in a floating point arithmetic". In: *Computational Geometry* 4 (1994). DOI: 10.1016/0925-7721(94)00017-4.

[125] R.A. Jarvis. "On the identification of the convex hull of a finite set of points in the plane". In: *Information Processing Letters* 2 (1973). DOI: 10.1016/0020-0190(73)90020-3.

[126] W. Jeon, G. Ko, J. Lee, H. Lee, D. Ha, and W.W. Ro. "Chapter Six - Deep learning with GPUs". In: *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*. Advances in Computers. 2021. DOI: 10.1016/bs.adcom.2020.11.003.

[127] D. Jiang and N.F. Stewart. "Backward Error Analysis in Computational Geometry". In: *Computational Science and Its Applications - ICCSA 2006*. 2006. DOI: 10.1007/11751540_6.

[128] J.R. Johnson, R.W. Johnson, D. Rodriguez, and R. Tolimieri. "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures". In: *Circuits, Systems and Signal Processing* 9 (1990).

[129] J. Jung, D. Park, G. Jo, J. Park, and J. Lee. "SnuRHAC: A Runtime for Heterogeneous Accelerator Clusters with CUDA Unified Memory". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 2021. DOI: 10.1145/3431379.3460647.

[130] Jaewoon Jung, Chigusa Kobayashi, Toshiyuki Imamura, and Yuji Sugita. "Parallel implementation of 3D FFT with volumetric decomposition schemes for efficient molecular dynamics simulations". In: *Computer Physics Communications* 200 (2016). ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.10.024.

[131] C.-D. Kang. "Measuring the effects of street network configurations on walking in Seoul, Korea". In: *Cities* 71 (2017). DOI: 10.1016/j.cities.2017.07.005.

[132] A.W. Keep. "A nanopass framework for commercial compiler development". PhD thesis. Indiana University, 2012. URL: https://andykeep.com/pubs/dissertation.pdf.

[133] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems". In: *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. 1994. DOI: 10.5555/1267074.1267084.

[134] K.N. Khan, M. Hirki, T. Niemi, J.K. Nurminen, and Z. Ou. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3 (2018). DOI: 10.1145/3177754.

[135] J.M. Kim, Y.G. Kim, and S.W. Chung. "Stabilizing CPU Frequency and Voltage for Temperature-Aware DVFS in Mobile Devices". In: *IEEE Transactions on Computers* 64 (2015). DOI: 10.1109/TC.2013.188.

[136] J. von Kistowski, H. Block, J. Beckett, C. Spradling, K.-D. Lange, and S. Kounev. "Variations in CPU Power Consumption". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. 2016. DOI: 10.1145/2851553.2851567.

[137] M. Knap and P. Czarnul. "Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs". In: *The Journal of Supercomputing* 75 (2019).

[138] I. Kodukula, N. Ahmed, and K. Pingali. "Data-Centric Multi-Level Blocking". In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. 1997. DOI: 10.1145/258915.258946.

[139] T.J. Koopman. *Generating Distributed Code from Array Languages (artefact)*. DOI: 10.5281/zenodo.15879814.

[140] T.J. Koopman. *Quickhull is Usually Forward Stable (artefact)*. 2025. DOI: 10.5281/zenodo.15915305.

[141] T.J. Koopman. "The tensor product of Bulk Synchronous Parallel algorithms". MA thesis. Utrecht University, Utrecht, The Netherlands, 2022.

[142] T.J. Koopman and R.H. Bisseling. "Minimizing Communication in the Multidimensional FFT". In: *SIAM Journal on Scientific Computing* 45.6 (2023). DOI: 10.1137/22M1487242.

[143] T.J. Koopman, T. Henriksen, C.E. Oancea, A.N. Ziogas, A.W. Hsu, D. van Balen, S. Smeding, T. De Matteis, S.-B. Scholz, I.G. de Wolff, T.L. McDonell, and G. Keller. *CFAL-bench*. 2025. URL: https://github.com/diku-dk/CFAL-bench/.

[144] T.J. Koopman, S.-B. Scholz, and B. van Gastel. "Artifact of the paper: Partitioning In-Place on Massively Parallel Systems". In: *31st International European Conference on Parallel and Distributed Computing*. 2025. DOI: 10.5281/zenodo.15576891.

[145] T.J. Koopman, S.-B. Scholz, and B. van Gastel. "Partitioning In-Place on Massively Parallel Architectures". In: (2025). DOI: 0.1007/978-3-031-99872-0_7.

[146] Thomas Koopman and Jordy Aaldering. *vecqhull*. 2025. URL: https://github.com/thomas3494/vecqhull.

[147] R. Kosloff. "Time-Dependent Quantum-Mechanical Methods for Molecular Dynamics". In: *Journal of Physical Chemistry* 92 (1988).

[148] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. "The Stanford FLASH Multiprocessor". In: *Proceedings of the 21st Annual International Symposium on Computer Architecture*. 1994. DOI: 10.1145/191995.192056.

[149] M.D. Lam, E.E. Rothberg, and M.E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms". In: *SIGPLAN Not.* 26 (1991). DOI: 10.1145/106973.106981.

[150] C. Lameter. "NUMA (Non-Uniform Memory Access): An Overview: NUMA Becomes More Common Because Memory Controllers Get Close to Execution Units on Microprocessors." In: *Queue* 11 (2013). DOI: 10.1145/2508834.2513149.

[151] L. Lamport. "How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor". In: *IEEE Transactions on Computers* 46.07 (1997). DOI: 10.1109/12.599898.

[152] R. Landaverde, Z. Tiansheng, A.K. Coskun, and M. Herbordt. "An investigation of Unified Memory Access performance in CUDA". In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 2014. DOI: 10.1109/HPEC.2014.7040988.

[153] C. Leforestier, R. H. Bisseling, C. Cerjan, M. D. Feit, R. Friesner, A. Guldberg, A. Hammerich, G. Jolicard, W. Karrlein, H.-D. Meyer, N. Lipkin, O. Roncero, and R. Kosloff. "A Comparison of Different Propagation Schemes for the Time Dependent Schrödinger Equation". In: *Journal of Computational Physics* 94 (1991).

[154] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. "The Stanford Dash multiprocessor". In: *Computer* 25 (1992). DOI: 10.1109/2.121510.

[155] R.J. LeVeque. "Finite difference methods for differential equations". In: *Draft version for use in AMath* 585 (1998).

[156] K. Li. "IVY: a shared virtual memory system for parallel computing". In: *Proceedings of the International Conference on Parallel Processing* 2 (1988).

[157] Kai Li and Paul Hudak. "Memory coherence in shared virtual memory systems". In: *ACM Trans. Comput. Syst.* 7.4 (1989). DOI: 10.1145/75104.75105.

[158] N. Li and S. Laizet. "A highly scalable 2D decomposition library and FFT interface". In: *Proceedings Cray User Group 2010 Conference*. 2010.

[159] Zhenyu Li and Victor Milenkovic. "Constructing strongly convex hulls using exact or rounded arithmetic". In: *Proceedings of the Sixth Annual Symposium on Computational Geometry*. SCG '90. Berkley, California, USA, 1990. ISBN: 0897913620. DOI: 10.1145/98524.98577.

[160] T.M. Low, F.D. Igual, T.M. Smith, and E.S. Quintana-Orti. "Analytical Modeling Is Enough for High-Performance BLIS". In: *ACM Trans. Math. Softw.* 43 (2016). DOI: 10.1145/2925987.

[161] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. "Introduction to the HPC Challenge Benchmark Suite". In: *LBL Publications* (2004).

[162] T. Macht and C. Grelck. "SAC Goes Cluster: Fully Implicit Distributed Computing". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019. DOI: 10.1109/IPDPS.2019.00107.

[163] M. Mairandres, C. Trinitis, J. Tao, and C. Osendorfer. "ViSMI: Software Distributed Shared Memory for InfiniBand Clusters". In: *2013 IEEE 12th International Symposium on Network Computing and Applications*. 2004. DOI: 10.1109/NCA.2004.1347776.

[164] J. Makhoul. "A fast cosine transform in one and two dimensions". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28 (1980). DOI: 10.1109/TASSP.1980.1163351.

[165] J. Malcolm, P. Yalamanchili, C. McClanahan, V. Venugopalakrishnan, K. Patel, and J. Melonakos. "ArrayFire: a GPU acceleration platform". In: *Modeling and Simulation for Defense Systems and Applications VII*. Vol. 8403. 2012. DOI: 10.1117/12.921122.

[166] C. Maples and L. Wittie. "MERLIN. A superglue for multicomputer systems". In: *Digest of Papers Compcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*. 1990. DOI: 10.1109/CMPCON.1990.63656.

[167] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka. "MACC: An OpenACC transpiler for automatic multi-GPU use". In: *Supercomputing Frontiers: 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings 4*. 2018. DOI: 10.1007/978-3-319-69953-0_7.

[168] J.D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995).

[169] T.L. McDonell, M.T. Chakravarty, G. Keller, and B. Lippmeier. "Optimising purely functional GPU programs". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. 2013. DOI: 10.1145/2500365.2500595.

[170] A.C. McKellar and E.G. Coffman. "Organizing Matrices and Matrix Operations for Paged Memory Systems". In: *Commun. ACM* 12 (1969). DOI: 10.1145/362875.362879.

[171] S. Meeran and A. Share. "Optimum path planning using convex hull and local search heuristic algorithms". In: *Mechatronics* 7.8 (1997). DOI: 10.1016/S0957-4158(97)00033-0.

[172] G. Mei. "CudaChain: an alternative algorithm for finding 2D convex hulls on the GPU". In: *SpringerPlus* 5 (2016). DOI: 10.1186/s40064-016-2284-4.

[173] D. Merrill and M. Garland. *Single-pass parallel prefix scan with decoupled look-back*. Tech. rep. NVIDIA, 2016.

[174] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[175] M. Milakov and N. Gimelshein. *Online normalizer calculation for softmax*. 2018. arXiv: 1805.02867 [cs.PF]. URL: https://arxiv.org/abs/1805.02867.

[176] P.A. Milder, F. Franchetti, J.C. Hoe, and M. Püschel. *Discrete Fourier Transform Compiler: From Mathematical Representation to Efficient Hardware*. CSSI Technical Report #CSSI-07-01, Carnegie Mellon University. 2007.

[177] P. Munksgaard, S.L. Breddam, T. Henriksen, F.C. Gieseke, and C.E. Oancea. "Dataset Sensitive Autotuning of Multi-versioned Code Based on Monotonic Properties". In: *Trends in Functional Programming*. 2021. DOI: 10.1007/978-3-030-83978-9_1.

[178] P. Munksgaard, T. Henriksen, P. Sadayappan, and C.E. Oancea. "Memory Optimizations in an Array Language". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2022. DOI: 10.1109/SC41404.2022.00036.

[179] *MVAPICH Benchmark*. http://mvapich.cse.ohio-state.edu/benchmarks/. Accessed: 2023-08-02.

[180] S. Naffziger, J. Warnock, and H. Knapp. "SE2 when processors hit the power wall (or "when the CPU hits the fan")". In: *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.* 2005. DOI: 10.1109/ISSCC.2005.1493852.

[181] M.J. Narasimha and A.M. Peterson. "On the Computation of the Discrete Cosine Transform". In: *IEEE Transactions on Communications* 26 (1978). DOI: 10.1109/TCOM.1978.1094144.

[182] J. Nickolls, I. Buck, M. Garland, and K. Skadron. "Scalable parallel programming with CUDA". In: *ACM SIGGRAPH 2008 Classes*. 2008. DOI: 10.1145/1401132.1401152.

[183] A. Nicolaisen and M.A. Persson. "Implementing Single-Pass Scan in the Futhark Compiler". MA thesis. Department of Computer Science, Faculty of Science, University of Copenhagen, https://futhark-lang.org/student-projects/marco-andreas-scan.pdf, 2020. URL: https://futhark-lang.org/student-projects/marco-andreas-scan.pdf.

[184] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. UK: Cambridge university press, 2010. ISBN: 9781107002173.

[185] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. "Global Arrays: a portable "shared-memory" programming model for distributed memory computers". In: *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. 1994. DOI: 10.1109/SUPERC.1994.344297.

[186] NVIDIA. *nbody*. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/nbody. Commit b20346741. 2024.

[187] NVIDIA Corporation. *CUB*. Version 1.17.2. URL: https://github.com/nvidia/cccl.

[188] *Oak Ridge's exascale 'Frontier' system named world's most powerful supercomputer on Top500*. https://www.datacenterdynamics.com/en/news/oak-ridges-exascale-frontier-system-named-worlds-most-powerful-supercomputer-on-top500/. Accessed: 2023-08-02.

[189] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. May 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[190] D. Pekurovsky. "P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions". In: *SIAM Journal on Scientific Computing* 34 (2012). DOI: 10.1137/11082748X.

[191] M. Pippig. "PFFT - An Extension of FFTW to Massively Parallel Architectures". In: *SIAM Journal on Scientific Computing* 35 (2013).

[192] S. Plimpton, A. Kohlmeyer, P. Coffman, and P. Blood. *fftMPI, a library for performing 2d and 3d FFTs in parallel*. 2018.

[193] S. Plimpton, R. Pollock, and M. Stevens. "Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations". In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. 1997.

[194] D.T. Popovici, M.D. Schatz, F. Franchetti, and T.M. Low. "A Flexible Framework for Multidimensional DFTs". In: *SIAM Journal on Scientific Computing* 42 (2020). DOI: 10.1137/19M1288401.

[195] J. Prades, B. Varghese, C. Reaño, and F. Silla. "Multi-tenant virtual GPUs for optimising performance of a financial risk application". In: *Journal of Parallel and Distributed Computing* 108 (2017). DOI: 10.1016/j.jpdc.2016.06.002.

[196] F. P. Preparata and S. J. Hong. "Convex hulls of finite sets of points in two and three dimensions". In: *Commun. ACM* 20 (1977). DOI: 10.1145/359423.359430.

[197] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. 2007.

[198] Omni OpenMP Compiler Project. *NPB3.0-omp-C*. `https://github.com/benchmark-subsetting/NPB3.0-omp-C`. 2014.

[199] A. Pryor, Y. Yang, A. Rana, M. Gallagher-Jones, J. Zhou, Y. Hung Lo, G. Melinte, W. Chiu, J.A. Rodriguez, and J. Miao. "GENFIRE: A generalized Fourier iterative reconstruction algorithm for high-resolution 3D imaging". In: *Scientific Reports* 7 (2017). DOI: `10.1038/s41598-017-09847-1`.

[200] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE* 93 (2005). DOI: `10.1109/JPROC.2004.840306`.

[201] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. "Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing". In: *Communications of ACM* 61 (2017). DOI: `10.1145/3150211`.

[202] B. Ramesh, C.J. Ribbens, and S. Varadarajan. "Is It Time to Rethink Distributed Shared Memory Systems?" In: *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. 2011. DOI: `10.1109/ICPADS.2011.75`.

[203] D. Romik. *The surprising mathematics of longest increasing subsequences*. 4. Cambridge, UK: Cambridge University Press, 2015.

[204] J. Rosenberg. "Some misconceptions about lines of code". In: *Proceedings fourth international software metrics symposium*. 1997. DOI: `10.1109/METRIC.1997.637174`.

[205] P. Schaad, T. Ben-Nun, and T. Hoefler. "Boosting Performance Optimization with Interactive Data Movement Visualization". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. 2022. DOI: `10.1109/SC41404.2022.00069`.

[206] R. Schenck, O. Rønning, T. Henriksen, and C.E. Oancea. "AD for an Array Language with Nested Parallelism". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2022. DOI: `10.1109/SC41404.2022.00063`.

[207] S.-B. Scholz. "Single Assignment C — Efficient Support for High-level Array Operations in a Functional Setting". In: *Journal of Functional Programming* 13 (2003). DOI: `10.1.1.138.6995`.

[208] S.-B. Scholz. "Single-assignment C — Functional Programming Using Imperative Style". In: *6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England, UK*. 1994.

[209] S.-B. Scholz. "Why rank-polymorphism matters". In: *22. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. 2023. DOI: `10.18154/RWTH-2023-10034`.

[210] S.-B. Scholz and A. Šinkarovs. "Tensor comprehensions in SaC". In: *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. 2021. DOI: 10.1145/3412932.3412947.

[211] S.B. Scholz. "With-loop-folding in Sac — Condensing Consecutive Array Operations". In: *Implementation of Functional Languages, 9th International Workshop*. 1998. DOI: 10.1007/BFb0055425.

[212] S. Schrijvers, T. Koopman, and S.-B. Scholz. "Shray: An Owner-Compute Distributed Shared-Memory System". In: *Proceedings of the 10th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2024. DOI: 10.1145/3652586.3663314.

[213] T. Schrijvers, S. Peyton Jones, M.T. Chakravarty, and M. Sulzmann. "Type checking with open type functions". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. 2008. DOI: 10.1145/1411204.1411215.

[214] P. Shamis, M. Gorentla Venkata, M.G. Lopez, M.B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R.L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. "UCX: An Open Source Framework for HPC Network APIs and Beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015. DOI: 10.1109/HOTI.2015.13.

[215] Y. Shiloach and U. Vishkin. "An O(n2log n) parallel max-flow algorithm". In: *Journal of Algorithms* 3 (1982). DOI: 10.1016/0196-6774(82)90013-X.

[216] *Shray*. https://gitlab.sac-home.org/sac-group/shray2. Accessed: 2023-08-02.

[217] *SHRAYonUCX*. https://gitlab.sac-home.org/sac-group/SHRAYonUCX. Accessed: 2025-07-10.

[218] S. Singh. "Comparative Study of Various Consistency Models in Distributed Shared Memory System". In: *The IUP Journal of Information Technology* 6 (2010).

[219] A. Šinkarovs, T.J. Koopman, and S.-B. Scholz. "Rank-Polymorphism for Shape-Guided Blocking". In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. 2023. DOI: 10.1145/3609024.3609410.

[220] A. Šinkarovs and S.-B. Scholz. "Parallel Scan as a Multidimensional Array Problem". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2022. DOI: 10.1145/3520306.3534500.

[221] A. Šinkarovs and S.-B. Scholz. "Parallel Scan as a Multidimensional Array Problem". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2022. DOI: 10.1145/3520306.3534500.

[222] A. Šinkarovs and S.-B. Scholz. "Portable Support for Explicit Vectorisation in C". In: *16th Workshop on Compilers for Parallel Computing (CPC'12)*. 2012.

[223] A. Šinkarovs, H.-N. Vießmann, and S.-B. Scholz. "Array languages make neural networks fast". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2021. DOI: 10.1145/3460944.3464312.

[224] S. Srungarapu, D.P. Reddy, K. Kothapalli, and P.J. Narayanan. "Fast Two Dimensional Convex Hull on the GPU". In: *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. 2011. DOI: 10.1109/WAINA.2011.64.

[225] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. "Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015. DOI: 10.1145/2784731.2784754.

[226] B.J. Svensson, M. Vollmer, E. Holk, T.L. McDonell, and R.R. Newton. "Converting data-parallelism to task-parallelism by rewrites: purely functional programs across multiple GPUs". In: *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*. 2015. DOI: 10.1145/2808091.2808093.

[227] P. Tang, L. Jiang, and H. Liu. "Collision Detection of High Density Point Set Based on Convex Hull". In: *2008 International Conference on Computer Science and Software Engineering*. Vol. 2. 2008. DOI: 10.1109/CSSE.2008.483.

[228] The CGAL Project. *CGAL User and Reference Manual*. 5.6.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/5.6.1/Manual/packages.html.

[229] N. Tollenaere, G. Iooss, S. Pouget, H. Brunie, C. Guillon, A. Cohen, P. Sadayappan, and F. Rastello. "Autotuning Convolutions Is Easier Than You Think". In: *ACM Transactions on Architecture and Code Optimization* 20 (2023). DOI: 10.1145/3570641.

[230] C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022). DOI: 10.1109/TPDS.2021.3097283.

[231] L. Trümper, T. Ben-Nun, P. Schaad, A. Calotoiu, and T. Hoefler. "Performance Embeddings: A Similarity-Based Transfer Tuning Approach to Performance Optimization". In: *Proceedings of the 37th International Conference on Supercomputing*. 2023. DOI: 10.1145/3577193.3593714.

[232] L.G. Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990).

[233] C. Van Loan. *Computational frameworks for the fast Fourier transform*. USA: Society for Industrial and Applied Mathematics, 1992. ISBN: 0898712858. DOI: 10.1137/1.9781611970999.

[234] F.G. Van Zee and R.A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software* 41 (2015). DOI: 10.1145/2764454.

[235] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[236] A. Vedaldi and K. Lenc. "MatConvNet: Convolutional Neural Networks for MATLAB". In: *Proceedings of the 23rd ACM International Conference on Multimedia*. 2015. DOI: 10.1145/2733373.2807412.

[237] M. Verloop, T.J. Koopman, and S.-B. Scholz. "Modulo in high-performance code: strength reduction for modulo-based array indexing in loops". In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. 2024. DOI: 10.1145/3652561.3652573.

[238] H.-N. Vießmann and S.-B. Scholz. "Effective Host-GPU Memory Management Through Code Generation". In: *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*. 2021. DOI: 10.1145/3462172.3462199.

[239] H. A. van der Vorst. "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 13 (1992). DOI: 10.1137/0913035.

[240] P. Wadler. "Deforestation: transforming programs to eliminate trees". In: *Theoretical Computer Science* 73 (1990). DOI: 10.1016/0304-3975(90)90147-A.

[241] David W Walker and Jack J Dongarra. "MPI: a standard message passing interface". In: *Supercomputer* 12 (1996). DOI: 10.1145/169627.169855.

[242] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013. DOI: 10.1145/2503210.2503219.

[243] R.C. Whaley, A. Petitet, and J.J. Dongarra. "Automated empirical optimizations of software and the ATLAS project". In: *Parallel Computing* 27 (2001). DOI: 10.1016/S0167-8191(00)00087-9.

[244] M. Wolfe. "More iteration space tiling". In: *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. 1989. DOI: 10.1145/76263.76337.

[245] A.N. Yzelman, R.H. Bisseling, D. Roose, and K. Meerbergen. "MulticoreBSP for C: a high-performance library for shared-memory parallel programming". In: *International Journal of Parallel Programming* 42 (2014).

[246] F.G. Van Zee, P. Bientinesi, T. M. Low, and R.A. van de Geijn. "Scalable Parallelization of FLAME Code via the Workqueuing Model". In: *ACM Trans. Math. Softw.* 34 (2008). DOI: 10.1145/1326548.1326552.

[247] Y. Zhang and F. Mueller. "CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures". In: *Proceedings of the 2012 41st International Conference on Parallel Processing*. ICPP'12. 2012. DOI: 10.1109/ICPP.2012.21.

[248] L. Zheng, C. Jia, M. Sun, Z. Wu, C. Hao Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J.E. Gonzalez, and I. Stoica. "Ansor: Generating High-Performance Tensor Programs for Deep Learning". In: *14th USENIX Symposium on Operating Systems Design and Implementation*. 2020. DOI: 10.5555/3488766.3488815.

[249] H. Zhou, Y. Mhedheb, K. Idrees, C.W. Glass, J. Gracia, and K. Fürlinger. "DART-MPI: An MPI-Based Implementation of a PGAS Runtime System". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 2014. DOI: 10.1145/2676870.2676875.

[250] A.N. Ziogas, T. Ben-Nun, G.I. Fernández, T. Schneider, M. Luisier, and T. Hoefler. "A Data-Centric Approach to Extreme-Scale Ab initio Dissipative Quantum Transport Simulations". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*. 2019. DOI: 10.1145/3295500.3357156.

[251] A.N. Ziogas, G. Kwasniewski, T. Ben-Nun, T. Schneider, and T. Hoefler. "Deinsum: Practically I/O Optimal Multilinear Algebra". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. 2022. DOI: 10.5555/3571885.3571918.

[252] A.N. Ziogas, T. Schneider, T. Ben-Nun, A. Calotoiu, T. De Matteis, J. de Fine Licht, L. Lavarini, and T. Hoefler. "Productivity, portability, performance: data-centric Python". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021. DOI: 10.1145/3458817.3476176.

# Summary

Computers are becoming increasingly complex. Clock frequencies no longer increase significantly, which forces hardware designers to design systems where multiple processors or execution units work together. There are multiple ways to design these systems, and each come with their strengths and weaknesses. Consequently, we now have many hardware architectures, that each come with their own programming environment. Maintaining an implementation for different architectures is a duplication of effort, and may become obsolete when new hardware hits the market. Instead, we can specify our implementations in a language that has no notion of a particular architecture, so that the compiler can generate code for multiple architectures. If the compiler cannot make a hardware-specific optimisation, this may lead to a loss of performance, but this dissertation argues that in most cases, optimisations can be made hardware independently. For that reason, the central thesis is that *it is time to abstract away hardware architecture in scientific computing.*

This dissertation is divided in two parts. The first part investigates to what extent hardware-agnostic languages can generate performant code. We first look at a wide range of applications and programming languages, and the code it generates for CPUs and GPUs. Overall, we obtain good results, but for a combinatorial problem called Quickhull, no language can come near the state of the art. We optimize the key component of this benchmark by hand using the same idea for GPUs and CPUs. This shows the idea is portable across hardware, even though current technology cannot generate this code automatically. We then contribute to code generation for distributed-memory machines, which extends portable performance from CPUs and GPUs, to systems with interconnected collections of them.

The second part develops theory on how to do portable performance engineering with higher level languages. The increased level of abstraction allows for new ways of structuring code, notably by using *rank-polymorphism.* The rank of an array is also known as its dimension, or the number of indices required to describe an element. A function is rank-polymorphic when it can be applied to arrays of any rank. We describe applications in matrix multiplication, the Fourier transform, and Quickhull. Interestingly, all three applications are substantially different. For matrix multiplication, we use rank-polymorphism to describe recursion, for the Fourier transform, we use it to understand multi-linearity, and for Quickhull we use it to control numerical error.

We conclude that it is possible to obtain performant code from a hardware-

agnostic specification for benchmarks in particle simulation and dense linear algebra. We see that irregular sparse linear algebra and combinatorial problems are challenging for array languages, and identify opportunities for future work in these areas.

# Samenvatting

Computers worden steeds complexer. Klok frequenties worden nauwelijks hoger, wat hardware ontwerpers dwingt om systemen te ontwerpen waar verschillende processoren samenwerken. Er zijn meerdere manieren om deze systemen te ontwerpen, die elk met hun sterke en zwakke kanten komen. Hierom hebben we nu vele hardware architecturen, die elk met hun eigen programmeeromgeving komen. Een implementatie voor elke architectuur onderhouden, is dubbele moeite, en wellicht snel achterhaald door nieuwe hardware. In plaats hiervan, kunnen we onze implementaties specificeren in een taal die geen besef heeft op welke hardware het draait. De compiler kan dan code genereren voor verschillende stukken hardware. Als de compiler geen hardware-specifieke optimalisaties kan maken, kan dit tot een prestatieverlies leiden. Echter, dit proefschrift beargumenteert dat in de meeste gevallen optimalisaties onafhankelijk van de hardware gemaakt kunnen worden. Daarom is de stelling die centraal staat in dit proefschrift dat *het is tijd om hardware architectuur weg te abstraheren in wetenschappelijke berekeningen.*

Dit proefschrift is in twee delen verdeeld. Het eerste deel onderzoekt in hoeverre hardware-onwetende talen snelle code kunnen genereren. We bekijken eerst een breed scala aan toepassingen en programmeertalen, en de code die voor CPUs en GPUs wordt gegenereerd. Over het algemeen behalen we goede resultaten, maar voor een combinatorische probleem dat Quickhull heet, kan geen taal in de buurt komen bij de stand van techniek. We optimaliseren het belangrijkste onderdeel van deze benchmark handmatig met hetzelfde idee voor zowel GPUs als CPUs. Dit laat zien dat het idee hardware kan overbruggen, alhoewel huidige technologie deze code nog niet kan genereren. We dragen daarna bij aan code voor gedistribueerd-geheugen machines, die overdraagbare prestaties uitbreidt van CPUs en GPUs naar verbonden verzamelingen daarvan.

Het tweede deel ontwikkelt theorie over hoe we code in abstracte talen kunnen schrijven die snel is op verscheidene soorten hardware. Het verhoogde abstractiegehalte maakt nieuwe manieren om code te structureren mogelijk, in het bijzonder *rang-polymorfisme*. De rang van een reeks staat ook bekend als de dimensie, of het aantal getallen om een index te beschrijven. Een functie is rang-polymorf wanneer het toegepast kan worden op reeksen van willekeurige rang. We beschrijven toepassingen in matrix-vermenigvuldiging, Fourier transformaties, en Quickhull. Interessant genoeg, zijn alle drie toepassingen wezenlijk verschillend. Voor matrix-vermenigvuldiging gebruiken we rang-polymorfisme om recursie te beschrijven, voor de Fourier transformatie om multi-lineariteit te begrijpen, en in Quickhull

om numerieke fouten te controleren.

We concluderen dat het mogelijk is om snelle code te genereren vanuit een taal die geen benul heeft van de hardware voor toepassingen in het beschrijven van deeltjes en lineaire algebra. We zien dat ijle lineaire algebra en combinatorische problemen een uitdaging zijn voor reeks-talen, en identificeren een aantal mogelijkheden voor verder werk in deze onderzoeksgebieden.

# Acknowledgement

Writing this thesis would not have been possible without the support of many others, both academic and non-academic.

First, I thank my promotor and daily supervisor Sven-Bodo Scholz. I have learned a lot from you about the importance of communicating or 'selling' ideas, and working carefully. Thank you for your patience and persistence in this, because these areas did not come naturally to me. I also thank my copromoters Bernard van Gastel, Pieter Koopman, and Peter Achten for the many brainstorming and feedback sessions.

To my colleagues of the software science team—my promotion team, Jordy Aaldering, Niek Janssen, Mart Lubbers, John van Groningen, Benedikt Rips and Saied Akbari Bibihayat—I offer the highest praise possible from a unionised worker: you have made this PhD four years of a 9–5 job instead of an arduous journey. Creating such an environment is not easy in academia, and I appreciate it. I also thank my less direct colleagues, Hilde Wolvers-van Raaij, Ingrid Berenbroek, Ben Polman, Simon Oosthoek, and Eric Lieffers for their part in keeping the department running. Gratitude to my many co-authors: doing science alone is boring. I would not be half the researcher I am today without everything I have learned from you. To the manuscript committee—Robbert Krebbers, Ana-Lucia Varbanescu, and Rob van Nieuwpoort—for reading this work carefully. I thank Rob H. Bisseling for introducing me to parallel computing and computer science in general.

Thank you to my friends and weekend colleagues: Jeff, Kelsey, Chris, Natasja, Asen, Michelle, Aiko, Patrick, Melanie, Shilpa, and so many others. I thank everyone in the VSA for making me feel welcome when I first moved to Nijmegen, the potlucks, and other activities.

Finally, I would like to thank my family; my parents, sister, grandmothers and other family members for their love and support.

# Research Data Management

This thesis research has been carried out under the research data management policy of the Institute for Computing and Information Science of Radboud University.[1]

The following research datasets have been produced during this PhD research:

- Chapter 2: Comparing Functional Array Languages.
  C, C++, DaCe, SaC, APL, Futhark, Accelerate code, available on
  `gitlab.com/software-technology/CFAL-bench`
  This is a preliminary version. The article is under review, and the benchmarks may be updated.

- Chapter 3: Partitioning In-Place.
  CUDA-code in a Git repository, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.15576891`

- Chapter 4: VQhull: a Fast Planar Quickhull.
  C++-code in a Git repository, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.15222744`

- Chapter 5: Shray: An Owner-Compute Distributed Shared-Memory System.
  C, Chapel, UPC, and Fortran code, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.15908751`

- Chapter 6: Distributed-Memory Code Generation.
  SaC code, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.15879814`

- Chapter 7: Multi-GPU Code Generation.
  SaC code, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.17036308`

- Chapter 9: Minimizing Communication in the Multidimensional FFT.
  C-code in a Git repository, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.15223245`

[1] `https://www.ru.nl/en/institute-for-computing-and-information-sciences/research`, bottom of page, last accessed April 15th, 2025

- Chapter 10: Shape-Guided Matrix Multiplication.
  SaC and C code, permanently available at ACM.
  `https://dl.acm.org/do/10.1145/3580402`
- Chapter 11: Quickhull is Usually Forward Stable.
  C code, permanently available on Zenodo.
  `https://doi.org/10.5281/zenodo.15915305`

# Curriculum Vitae

Thomas Koopman

| | |
|---|---|
| 1998 | Born July 6th, Leidschendam |
| 2010–2016 | VWO at Sint-Maartenscollege, Voorburg |
| 2016–2019 | Bachelor's degree (cum laude) Mathematics at the Utrecht University, Utrecht |
| 2019–2022 | Master's degree (cum laude) Mathematical Sciences at the Utrecht University, Utrecht |
| 2022–2026 | PhD candidate at the Radboud University, Nijmegen in the Institute for Computing and Information Sciences (iCIS). |


Radboud Universiteit