UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Troels Henriksen — athas@sigkill.dk

Design and Implementation of the Futhark Programming Language (Revised)

Supervisors: Cosmin Eugen Oancea and Fritz Henglein

December, 2017

Abstract

In this thesis we describe the design and implementation of Futhark, a small data-parallel purely functional array language that offers a machine-neutral programming model, and an optimising compiler that generates efficient OpenCL code for GPUs. The overall philosophy is based on seeking a middle ground between functional and imperative approaches. The specific contributions are as follows:

First, we present a *moderate flattening* transformation aimed at enhancing the degree of parallelism, which is capable of exploiting easily accessible parallelism. Excess parallelism is efficiently sequentialised, while keeping access patterns intact, which then permits further locality-of-reference optimisations. We demonstrate this capability by showing instances of automatic loop tiling, as well as optimising memory access patterns.

Second, to support the flattening transformation, we present a lightweight system of size-dependent types that enables the compiler to reason symbolically about the size of arrays in the program, and that reuses general-purpose compiler optimisations to infer relationships between sizes.

Third, we furnish Futhark with novel parallel combinators capable of expressing efficient sequentialisation of excess parallelism, as well as their fusion rules.

Fourth, in order to express efficient programmer-written sequential code inside parallel constructs, we introduce support for safe in-place updates, with type system support to ensure referential transparency and equational reasoning.

Fifth, we perform an evaluation on 21 benchmarks that demonstrates the impact of the language and compiler features, and shows application-level performance that is in many cases competitive with hand-written GPU code.

Sixth, we make the Futhark compiler freely available with full source code and documentation, to serve both as a basis for further research into functional array programming, and as a useful tool for parallel programming in practice.

Resumé

Denne afhandling beskriver udformningen og implementeringen af Futhark, et enkelt data-parallelt, sideeffekt-frit, og funktionsorienteret geledsprog, der frembyder en maskinneutral programmeringsmodel. Vi beskriver ligeledes den tilhørende optimerende Futhark-oversætter, som producerer effektiv OpenCL-kode målrettet afvikling på GPUer. Den overordnede designfilosofi er at ydnytte både functionsorienterede og imperative fremgangsmåder. Vores konkrete bidrag er som følger:

For det første præsenterer vi en moderat fladningstransformering, der er i stand til at udnytte blot den grad af parallelisme som er nødvendig eller lettilgængelig, og omdanne overskydende parallelisme til effektiv sekventiel kode. Denne sekventielle kode bibeholder oprindelig lageradgangsmønsterinformation, hvilket tillader yderligere lagertilgangsforbedringer. Vi demonstrerer nytten af denne egenskab ved at give eksempler på automatisk blokafvikling af løkker, samt ændring af lageradgangsmønstre således at GPUens lagersystem udnyttes bedst muligt.

For det andet beskriver vi, med henblik på understøttelse af fladningstransformeringen, et enkelt typesystem med størrelses-afhængige typer, der tillader oversætteren at ræsonnere symbolsk om størrelsen på geledder i programmet under oversættelse. Vores fremgangsmåde tillader genbrug af det almene repertoire af oversætteroptimeringer i spørgsmål om ligheder mellem størrelser.

For det tredje udstyrer vi Futhark med en række nyskabedne parallelle kombinatorer der tillader effektiv sekventialisering af unødig parallelisme, samt disses fusionsregler.

For det fjerde indfører vi, med henblik på at understøtte effektiv sekventiel kode indlejret i de parallelle sprogkonstruktioner, understøttelse for direkte ændringer i geledværdier. Denne understøttelse sikres af et typesystem der garanterer at effekten ikke kan observeres, og at lighedsbaseret ræsonnering stadigvæk er muligt.

For det femte foretager vi en ydelsessammenlining indeholdende 21 programmer, med henblik på at demonstrere sprogets praktiske anvendelighed og oversætteroptimeringernes indvirkning. Vores resultater viser at Futhark's overordnede ydelse i mange tilfælde er konkurrencedygtig med håndskreven GPU-kode.

For det sjette gør vi Futhark-oversætteren frit tilgængelig, inklusive al kildetekst og omfattende dokumentation, således at den kan tjene både som et udgangspunkt for yderligere forskning i funktionsorienteret geledprogrammering, samt som et praktisk andvendeligt værktøj til parallelprogrammering.

Contents

Preface

This thesis is submitted in fulfillment of the PhD programme in computer science (*Datalogi*) at the University of Copenhagen, for Troels Henriksen, under the supervision of Cosmin Eugen Oancea and Fritz Henglein.

Publications

Of the peer-reviewed papers I have published during my studies, the following contribute directly to this thesis:

> HENRIKSEN, TROELS, MARTIN ELSMAN, and COSMIN E OANCEA. "Size slicing: a hybrid approach to size inference in Futhark". In: *Proc. of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*. ACM. 2014, pp. 31–42

> HENRIKSEN, TROELS, KEN FRIIS LARSEN, and COSMIN 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*. ARRAY 2016. Santa Barbara, CA, USA: ACM, 2016, pp. 17–24

> HENRIKSEN, TROELS, NIELS GW SERUP, MARTIN ELSMAN, FRITZ HENGLEIN, and COSMIN 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*. ACM. 2017, pp. 556–571

While the following do not:

> HENRIKSEN, TROELS and COSMIN E OANCEA. "Bounds checking: An instance of hybrid analysis". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM. 2014, p. 88

HENRIKSEN, TROELS, MARTIN DYBDAL, HENRIK URMS, ANNA SOFIE KIEHN, DANIEL GAVIN, HJALTE ABELSKOV, MARTIN ELSMAN, and COSMIN OANCEA. "APL on GPUs: A TAIL from the Past, Scribbled in Futhark". In: *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*. FHPC'16. Nara, Japan: ACM, 2016, pp. 38–43

LARSEN, RASMUS WRIEDT and TROELS HENRIKSEN. "Strategies for Regular Segmented Reductions on GPU". in: *Proceedings of the 6th ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '17. New York, NY, USA: ACM, 2017

Acknowledgements

Language design and compiler hacking easily becomes a lonely endeavour. I would in particular like to thank Cosmin Oancea, Martin Elsman, and Fritz Henglein for being willing to frequently and vigorously discuss language design issues. I am also grateful to all students who contributed to the design and implementation of Futhark. In no particular order: Niels G. W. Serup, Chi Pham, Maya Saietz, Daniel Gavin, Hjalte Abelskov, Mikkel Storgaard Knudsen, Jonathan Schrøder Holm Hansen, Sune Hellfritzsch, Christian Hansen, Lasse Halberg Haarbye, Brian Spiegelhauer, William Sprent, René Løwe Jacobsen, Anna Sofie Kiehn, Henrik Urms, Jonas Brunsgaard, and Rasmus Wriedt Larsen. Development of the compiler was aided by Giuseppe Bilotta's advice on OpenCL programming, and James Price's work on `oclgrind` [PM15]. I am also highly appreciative of the work of all who contributed code or feedback to the Futhark project on Github[1], including (but not limited to): David Rechnagel Udsen, Charlotte Tortorella, Samrat Man Singh, `maccam912`, `mrakgr`, Pierre Fenoll, and `titouanc`

I am also particularly grateful for Cosmin Oancea's choice to devise a research programme that has allowed me the opportunity to spend three years setting up GPU drivers on Linux. And relatedly, I am grateful to NVIDIA for donating the K40 GPU used for most of the empirical results in this thesis.

And of course, I am grateful that Lea was able to put up with me for these three and a half years, even if I still do not understand why.

For this revised version of my thesis, I am also grateful to the feedback of my assessment committee—Torben Mogensen, Mary Sheeran, and Alan Mycroft—whose careful reading resulting in many improvements to the text.

The cover image depicts the common European hedgedog (*Erinaceus europaeus*), and is from *Iconographia Zoologica*, a 19th century collection of zoological prints. Like hedgehogs, functional languages can be much faster than they might appear.

[1] `https://github.com/diku-dk/futhark`

Part I

Land, Logic, and Language

Chapter 1

Introduction

This thesis describes the design and implementation of *Futhark*, a data parallel functional array language. Futhark, named after the first six letters of the Runic alphabet, is a small programming language that is superficially similar to established functional languages such as OCaml and Haskell, but with restrictions and extensions meant to permit compilation into efficient parallel code. While this thesis contains techniques that could be applied in other settings, Futhark has been the overarching context for our work. We demonstrate the applicability and efficiency of the techniques by their application in the Futhark compiler, and the performance of the resulting code. Apart from serving as a vehicle for compiler research, Futhark is also a programming language that is useful in practice for high-performance parallel programming.

It is the task of the compiler to map the high-level portable parallelism to low-level parallel code that can be executed by some machine. For example, a parallel language may support *nested* parallelism, while most machines efficiently support only *flat* parallelism. One problem is that not all parallelism is born equal, and the optimal depth of parallelisation depends not just on the static structure of the program, but also on the problem size encountered at run-time. When writing a program in a data parallel language, the program tends to contain a large amount of parallelism; frequently much more than what is necessary to fully exploit the target machine. For example, the program may contain two parallel loops, where executing the second loop in parallel carries significant overhead. If the outer loop contains enough parallel iterations to fully saturate the machine, then it is better to execute the inner loop sequentially. On the other hand, if the outer loop contains comparatively few iterations, then it may be worth paying the overhead of also executing the innermost loop in parallel. This decision cannot be made at compile-time, and thus the compiler should produce code for both versions, and pick between them at run-time, based on the problem size.

Many implementation techniques for data-parallelism have been developed. From the field of functional programming, we have Guy Blelloch's seminal *full flattening algorithm* [Ble+94], which shows how to transform *all* parallelism in a program writ-

ten in a nested data-parallel language to flat parallelism. While full flattening is guaranteed to preserve the asymptotic cost and parallelism of the program, its practical performance is often poor, because some of the parallelism may carry heavy overhead to exploit, and is unnecessary in practice. An example is parallel code hidden behind branches. Furthermore, the code generated by full flattening has an un-analysable structure, preventing locality-of-reference optimisations such as loop tiling.

For these reasons, flattening is not used much in practice. Instead, we find techniques that tend to exploit only easily accessible parallelism. One common approach is to exploit only top-level parallelism, with nested parallelism exploited only in limited ways, for example by always mapping it to a specific layer of the parallel hierarchy of the machine [GS06; Lee+14; McD+13; Ste+15]. While the resulting code tends to exhibit little overhead, the lack of flexibility makes it hard to express algorithms whose natural expression contains nested parallelism. This is particularly problematic when exploiting the inner levels of parallelism is critical to obtaining performance in practice.

More powerful techniques exist; for example, dynamic techniques for detecting parallelism [OM08] can in principle exploit also partial parallelism, but at significant run-time overhead. Static analyses such as the polyhedral model [Pou+11; RKC16; Ver+13; CSS15] are able to support some forms of nested parallelism, but are typically limited to affine code, and their success is not guaranteed by the language semantics.

In this thesis, we bridge the imperative and functional approaches: we seek to exploit only that parallelism which is necessary or easily accessible, and we work on the foundation of an explicitly parallel language. We are inspired by the systematic approach of flattening, but we wish to employ full flattening only as a last resort, when absolutely all parallelism must be exploited. Our goal is pragmatic. We wish to write Futhark programs in a high-level hardware-agnostic and modular style, using nested parallelism where convenient. The compiler should then translate the Futhark programs to low-level GPU code whose run-time performance rivals that of hand-written code.

First Contribution Our key contribution is *moderate flattening* (Chapter 8). This algorithm, which is inspired by both full flattening and loop distribution, is capable of restricting itself to exploiting only that parallelism which is cheap to access, or more aggressively exploit more of the available parallelism. Excess parallelism is turned into efficient sequential code, with enough structure retained to perform important locality-of-reference optimisations (Chapter 9). The moderate flattening algorithm is also capable of improving the degree of available parallelism by rewriting the program. For example, loop interchange is used to move parallel loop levels next to each other, even if they were separated by a sequential loop in the source program.

The moderate flattening algorithm is capable of data-sensitive parallelisation, where different parallelisations are constructed for the same program, and the optimal one picked at run-time.

Second Contribution The moderate flattening algorithm requires significant infrastructure to function. While full flattening transforms everything into flat bulk parallel constructs, within which all variables are scalars, we wish to permit just partial parallelisation. As a result, some variables in the sequentialised excess parallelism may be arrays. The compiler must have a mechanism for reasoning about their sizes.

One contribution here is a technique for *size inference* on multidimensional arrays (Chapter 6) that gives us the property that for every array in the program, there is a variable (or constant) in scope that describes the size of each dimension of the array. This allows the compiler to reason about the variance of arrays in a symbolic and simple way. Prior work tends to either restrict the language to ensure that all symbolic sizes can be statically determined [Jay99], or requires extensive annotation by the programmer [Bra13; BD09]. Our technique for size inference is built on a simple form of existential types. It is a pragmatic design that does not restrict the language (by forbidding for example filtering), but also does not require any annotations on behalf of the programmer, although it can benefit from such.

Third Contribution We present an extension to existing work on fusion (Chapter 7). The primary novelty is the creation of new array combinators that permit a greater degree of fusibility. Because of the moderate flattening technique, the fusion algorithm cannot know whether some piece of code will eventually be parallelised or sequentialised. We present constructs that have good fusion properties, and allow simultaneously exploitation of all parallelism, as well transformation into efficient sequential code. These are not properties provided by prior work on fusion. We also show how "horizontal" fusion (fusion between two loops that have no dependencies on each other) can be considered a special case of "vertical" (producer/consumer) fusion, given sufficiently powerful combinators, and can be an enabler of vertical fusion.

Fourth Contribution Even in a parallel language, it is sometimes useful to implement efficient sequential algorithms, often applied in parallel to parts of a data set. Most functional parallel languages do not support in-place updates at all, which hinders the efficient implementation of such sequential algorithms. While imperative languages obviously support in-place updates efficiently, they do not guarantee safety in the presence of parallelism.

We present a system of uniqueness types (Section 2.5.1), and a corresponding formalisation (Section 5.3), that permits a restricted form of in-place updates, that provides the cost guarantees without compromising functional purity or parallel se-

mantics. While more powerful uniqueness type systems [BS93], and affine and linear types [TP11; FD02] are known, ours is the first application that directly addresses `map`-style parallel constructs, and shows how in-place updates can be supported without making evaluation order observable. We show that the addition of uniqueness types does not overly burden the implementation of compiler optimisations.

Fifth Contribution We demonstrate the GPU performance of code generated by the Futhark compiler on a a set of 21 nontrivial problems adapted from published benchmark suites. 16 of the benchmarks are compared to hand-written implementations, but we also include five programs ported from Accelerate [McD+13], a mature Haskell library for parallel array programming. Our results show that the Futhark compiler generates code that performs comparably with hand-written code in most cases, and generally outperforms Accelerate. The results indirectly validate the low overhead of size analysis, and also show the impact of both uniqueness types and the various optimisations performed by the Futhark compiler.

Sixth Contribution The Futhark compiler is implemented with approximately 45, 000 lines of Haskell, and is available under a free software licence at

`https://github.com/diku-dk/futhark/`

The compiler and language is sufficiently documented for usage by third parties. The compiler performs several operations whose implementations took significant engineering effort, but are not covered in this thesis due to lack of scientific contribution. These include defunctorisation for an ML-style higher order module system, hoisting, variance/data-dependency analysis, memory management, memory expansion, OpenCL code generation, and a plethora of classical optimisations.

We believe that the Futhark compiler can serve as a starting point for future research in optimising compilers for explicitly parallel languages, as well as a tool for implementing data-parallel algorithms.

Closing At a high level, we argue that techniques from imperative compilers can be lifted to a functional and explicitly parallel setting, where analyses are driven by high-level properties of language constructs, rather than complex and low-level analysis, and hence prove more robust. We prove that using these techniques, a compiler can be written that generates highly performant code for non-contrived programs. We demonstrate the resulting performance compared to hand-written code (Chapter 10), where Futhark in most cases approaches or even exceeds the performance of hand-written low-level code.

Most of this work has previously been published in FHPC 2013 [HO13] (during the author's master's studies), FHPC 2014 [HEO14], ARRAY 2016 [HLO16] and PLDI 2017 [Hen+17].

Chapter 2

Background and Philosophy

In physics, the speed of light, denoted c, is the ultimate speed limit of the universe. Likewise in programming, "as fast as C" is often used as an indication that some programming language is as fast as it can possibly be. Of course, in theory it makes no sense to say that a given programming language is "slow" or "fast", as these are merely properties of a particular implementation of the programming language running on a specific computer. But in practice, it is clear that the design of a programming language has an overwhelming influence on the ease of constructing a performant implementation.

For decades, the design of languages such as C has permitted the implementation of compilers that generate efficient code. One important reason is that the abstract machine model underlying C, a sequential random-access register machine, can be mapped easily to mainstream computers with little overhead. This means that a programmer can write code in C and have a reasonable idea of the performance of the resulting code. It also means that even a naive non-optimising C compiler can generate code with good performance, although it may require somewhat more care from the C programmer. Indeed, C has often been described as "portable assembly code".[1]

In contrast, languages whose abstract model is more different from physical machines, so-called "high-level languages", cannot be as easily mapped to real machines. Compilers must invest considerable effort in mapping for example the call-by-need lambda calculus of Haskell to the sequential register machine [Jon92]. Even after decades of work, high-level languages struggle to catch up to the performance of C on the sequential random-accesss register machine.

Of course, a cursory study of modern hardware designs show that the sequential random-access register machine is an illusion. The abstract model of C may match a 70s minicomputer reasonably well, but it is increasingly diverging from how modern computers are constructed. This is despite the efforts of hardware designers: due to the massive popularity (and therefore economic significance) of C-like languages,

[1] I will using C as the main point of reference in the following sections, but the problems I point out are common to any sequential language.

manufacturers have attempted to prolong the illusion of the sequential random-access machine for as long as possible.

Unfortunately, while C may be exceeded (as we hope to show in this thesis), c is a tougher nut to crack. The machine on which this paragraph is being typed contains a CPU with a clock frequency of $2.3GHz$, corresponding to a clock cycle taking $0.43ns$. Given that c is approximately $3 \cdot 10^8 m/s$, light moves approximately $13cm$ in the the time the CPU takes to perform a clock cycle, thus physically limiting the distance a single piece of information can travel in a single cycle.

Another, even greater, physical limitation is power dissipation: the power usage of the transistors in a processor is roughly proportional to the square of its clock frequency [DAS12]. This prevents an increase in clock frequency unless we can compensate with an increase in the efficiency of the transistors. For a sequential machine that executes instructions in exactly the order they are given, the only way to speed up execution of a program is to increase the rate at which instructions are executed. If we can double the rate at which instructions are processed, we in effect halve the time it takes to execute some program (assuming no memory bottlenecks, which we will discuss later).

In the popular vernacular, *Moore's Law* is typically rendered as "computers double in speed every two years". But actually, the law states that *transistor density* doubles every two years (implying that transistors become smaller). Moore's Law does not state that the enlargened transistor budget translates straightforwardly to improved performance, although that was indeed the case for several decades. This is due to another law, *Dennard scaling*, which roughly states that as transistors get smaller, their power density stays constant. Taken together, Moore's Law and Dennard scaling rougly say that we can put ever more transistors into the same physical size, and with the same power usage (and thus heat generation).

The reduction in power usage per transistor granted by Dennard scaling permitted a straightforward increase in CPU clock frequency. Roughly, every time we cut the power consumption of a transistor in half via shrinkage, we can increase the clock frequency by 50%. For a sequential computer, this translates into a 50% performance increase. We can partially circumvent the limits of c by using techniques such as *pipelining*. Instruction latency may not be decreased with pipelining, but throughput can increase significantly. While it is clear that Moore's Law will eventually stop, or else transistors would eventually be smaller than a single atom, this is not the issue that has hindered the illusion of the sequential machine.

The problem we now face is that Dennard scaling began to break down around 2006. Physical properties of the circuit material lead to increased current leakage at small sizes, causing the chip to heat up. Simply pushing up the clock frequency became no longer viable. Instead, chip manufacturers are now using their transistor budget on increasing the amount of work that can be done in a clock cycle through various means:

Increasing the size of caches. One significant problem with modern computer designs is that processors are significantly faster than the memory from which they get their data. As a rule of thumb, accessing main memory incurs a latency of $100ns$—likely hundreds of clock cycles on a current CPU. This so-called *memory wall* can be circumvented by the addition of caches that store small subsets of the larger memory [WM95]. Heuristics, usually based on temporal locality, are useed to determine which parts of the larger memory are stored in the caches. High-performance CPUs can use several layers of caches, each layer being larger and slower than the former.

Inferring instruction-level parallelism. While the semantics of a sequential CPU is that instructions are executed one after another, it is often the case that two instructions to not have dependencies on one another, and thus can be executed in parallel. CPUs even perform *out-of-order* execution where later instructions are executed before earlier ones, if the latter instruction has no dependence on the latter.

Explicit parallelism at the hardware level. The two former techniques try to masquerade the fact that the sequential machine model is increasingly physically untenable. Another approach is to explicitly provide programmers with hardware support for parallel execution. This most famously takes the form of adding additional CPU cores, each of which processes a sequential stream of instructions, but an equally important technique is to add *vector instructions* that operate on entire vectors of data, rather than single values. As an example, the newest Intel AVX-512 vector instruction set provides instructions that operate on entire 512-bit vectors (for example, 16 single precision floating point values). Such instructions dramatically increase the amount of work that is done in a single clock cycle.

These techniques are all based on merely extending and refining the classic CPU design. While code may have to be re-written to take advantage of certain new hardware features (such as multiple cores or vector instructions), the programming experience is not too dissimilar from programming sequential machines. This is not necessarily a bad thing: sequential machines are easy to reason about, generous in their support of complex control flow, and have a significant existing base of programmer experience.

However, the notion that C is "portable assembly" begins to crack noticeably even at this point. C itself has no built-in notion of multi-threaded programming, nor of vector operations, and thus vendors have to provide new programming APIs (or language extensions) for programmers to take advantage of the new (performance-critical) machine features. Alternatively, C compilers must perform significant static analysis of the sequential C code to find opportunities for vectorisation or multi-threading. In essence, the compiler must reverse engineer the sequential statements

written by the programmer to reconstruct whatever parallelism may be present in the original algorithm.

The problem is that C was carefully designed for a particularly class of machines; a machine no longer resembled by modern high-performance computers. Thus, the *impedance mismatch* between C and hardware continues to grow, with little sign of stopping. This mismatch becomes acute when we move away from the comparatively benign world of multicore CPUs and into the realm of massively parallel processors.

In this chapter we shall discuss the problems caused by the impedance mismatch, and what a better programming model may look like. We begin in Section 2.1 by giving an overview of massively parallel machines, in particular the now-ubiquitous GPUs. Section 2.2 presents data-parallel functional programming, a programming model suited for massively parallel machines, through *Futhark*, the programming language that has been developed as part of this thesis. In Sections 2.3 and 2.4 we argue for the importance of fusion and nested parallelism for modular parallel programming. In Section 2.5 we defend the decision to introduce yet another programming language, rather than re-use an existing one, by arguing that certain features of existing functional languages hinder efficient parallel execution. Section 2.6 shows how Futhark, a restricted data-parallel language, can be used in practice, by demonstrating interoperability with mainstream and not-so-mainstream technologies. Finally, Section 2.7 shows the difficulties involved in parallelising legacy sequential code, and Section 2.8 discusses other approaches to data-parallel functional programming.

2.1 Massively Parallel Machines

To a large extent, mainstream CPUs favour programming convenience, familiarity, and backwards compatibility over raw performance. We must look elsewhere for examples of the kind of hardware that could be built if we were willing to sacrifice our traditional notions of sequential programming.

For as long as there have been computers, there have been parallel computers. Indeed, if we think back to the very earliest computers, the human ones made of flesh and blood, parallelism was the *only* way to speed up a computation. You were unlikely to make a single human computer much faster through training, but you could always pack more of them into a room. However, with the rise of electrical computers, and in particular the speed with which sequential performance improved, parallelism dwindled into the background. Certainly, you could have two computers cooperate on solving some problem, but the programming difficulties involved made it more palatable to simply wait for a faster computer to enter the market. Only the largest computational problems were worth the pain of parallel programming.[2]

[2] *Concurrency* was alive and well, however, due to its importance in multiuser operating systems and multitasking in general. But most concurrent programs were executed on sequential machines, with the illusion of concurrency formed through multiplexing. Many of the techniques developed for concurrent programming are also applicable to parallel programming, although concurrency tends to

Several interesting parallel computers were designed for *high-performance computing* (HPC) applications. One of the earliest and most influential was the Cray-1 from 1976, which was the first vector processor. About 80 were sold, which was a large number for a high-end supercomputer. A different design was the CM-1 from 1985, which was based on a computer built from a large number of simple 1-bit microprocessors. The CM-1 proved difficult to program and was not a success in the market, but was an early example of a *massively parallel machine* [Hil89].

In the 1990s, consumer demand for increasing visual fidelity in video games led to the rise of the *graphics processing unit* (GPU), for accelerating graphical operations that would be too slow if executed on the CPU. Initially, GPUs were special-purpose non-programmable processors that could only perform fixed graphical operations. Over time, the need for more flexibility in graphical effects led to the development of programmable *pixel shaders*, first seen in the NVIDIA GeForce 3 in 2000. Roughly, pixel shaders allowed an effect to be applied to every pixel of an image—for example, looking at every pixel and adding a reflection effect to those that represent water. Graphical operations such as these tend to be inherently parallel, and as a result, GPU hardware evolved to support efficient execution of programs with very simple control flow, but a massive amount of fine-grained parallelism. Eventually, GPU manufacturers started providing APIs that allowed programmers to exploit the parallel computational power of the now-mainstream GPUs even for non-graphics workloads, so-called *general-purpose GPU programming* (GPGPU). The most popular such APIs are CUDA [17] from NVIDIA, and OpenCL [SGS10], which is an open standard maintained by the Khronos Group.

GPUs are not the only massively parallel machines in use. However, their great success, their availability, the difficulty in programming them, and their potential compute power makes them excellent objects of study for researchers of parallel programming models. In this work, we will focus on the details of GPUs over other parallel machines. However, a central thesis of the work is that modern hardware is too complicated and too diverse for low-level programming by hand to be viable. We will introduce a high-level hardware-agnostic programming model, in the form of the Futhark programming language, and describe its efficient mapping to GPUs. The motivation is that if the model can be mapped efficiently to a platform as restricted as GPUs, it can probably also be mapped to other parallel platforms, such as multicore CPUs, clusters, or maybe even FPGAs.

2.1.1 Basic Properties of GPUs

The performance characterictics and programming model of modern GPUs is covered in greater detail in Chapter 4, but a basic introduction is given here, in order to give an idea of the difficulties involved in retrofitting sequential languages for GPU execution.

implicitly focus on correctness over performance on some specific machine.

GPUs derive their performance from an execution model called *single instruction multiple thread* (SIMT), which is very similar to the *single instruction multiple data* model. Roughly, threads are not fully independent, but grouped into bundles that all execute the same operations on different parts of a large data set. For example, on an NVIDIA GPU, threads are bundled into *warps*, of 32 threads each, that execute in lockstep, and which form the unit of scheduling. This execution is highly suited for dense and regular computations, such as the ones found in linear algebra, but less so for irregular and branch-heavy computations such as graph algorithms.

A GPU is still subject to the laws of physics, and there is therefore a significant latency between issuing a memory read, and actually receiving the requested data (the memory wall). On a CPU, a hierarchy of caches is used to decrease the latency, but a GPU uses aggressive *simultaneous multithreading*, where a thread that is waiting for a memory operation to complete is de-scheduled and another thread run in its place. This scheduling is done entirely in hardware, and thus does not carry the usual overhead of context switches. This scheduling is not done on each thread in isolation, but on entire 32-thread warps. Thus, while a GPU may only have enough computational units (such as ALUs) to execute a few thousand parallel instructions per clock cycle, tens of thousands of threads may be necessary to avoid the computational units being idle while waiting for memory operations to finish. As a result of this design, memory can also be optimised for very high bandwidth at the expense of latency, and it is not unusual for GPU memory buses to support bandwidth in excess of $300 GiB/s$ (although as we shall discuss in Chapter 4, specific access patterns must be followed to reach this performance).

GPUs have many limitations that hinder traditional programming techniques and languages:

- GPUs function as *co-processors*, where code is explicitly uploaded and invoked. A GPU program is typically called a *GPU kernel*, or just *kernel*.
- The number of threads necessary implies that each thread can use only relatively few registers and little memory (including stack space).
- The lockstep execution of warps, as well as the very small per-thread stack size, prevents function pointers and recursion.
- GPUs cannot directly access CPU memory, and so data must manually be copied to and from the GPU.[3]
- The memory hierarchy is explicit. While a small amount of cache memory is often present, obtaining peak performance depends on judicious use of programmer-managed on-chip memory.

[3]This is changing with the newer generation of GPUs.

- Memory allocation is typically not possible while executing a GPU program. All necessary memory must be pre-computed before starting GPU execution.
- Generally no support for interrupts or signals.

The mismatch between the abstract machine model of C and GPU architectures is clear. While current GPU programming APIs do in fact use a restricted subset of C (or C++ in the case of CUDA) for expressing the per-thread code, it is the programmmer's responsibility to orchestrate their execution. This involves organising communication between the tens of thousands of threads that are needed to saturate the hardware, with both correctness and performance easily suffering under even small mistakes—a tall order, and the compiler offers little help. In practice, this renders GPU programming the domain of experts, and off-limits to most programmers, except through the use of restricted libraries offering pre-defined primitives.

2.2 A Parallel Programming Language

We need a better programming model; one that does not have as fundamental a mismatch between its execution model and parallel hardware. This does not mean we need a low-level language, or a specialised GPU language. Rather, we should *increase* the level of abstraction, and stop overspecifying details like iteration order and sequencing, unless we really *need* something to execute in some specific order. When writing a C program, the programmer writes the program statements in some order, and the C compiler must then perform significant work to determine which of the sequencing restrictions are essential, and which are accidental. In a (pure) functional language, only data dependencies affect ordering. Furthermore, the language should be safe for parallel execution by construction, without the risk of race conditions or deadlocks.

In the following, I will introduce the *Futhark programming language*, a purely functional ML-like array language that has been developed as part of my PhD work[4]. Futhark is statically typed and eagerly evaluated. The full Futhark language supports both parametric polymorphism and an advanced higher-order module system inspired by Standard ML [MTM97], but for this thesis we shall restrict ourselves to the simple monomorphic subset. Futhark is not a general-purpose programming language. Instead, Futhark is intended for relatively small and simple high-performance programs, which can form the computational core of a larger program written in a conventional language. This perspective is elaborated in Section 2.6. This thesis is not intended as a full guide to Futhark programming; the Futhark reference manual[5] serves that purpose well enough. Therefore, we explain the syntax and semantics

[4] Futhark has its roots in an earlier language, $\mathcal{L}_0$, which I helped develop during my master's studies.

[5] `https://futhark.readthedocs.io`

of Futhark on a case-by-case basis by example. We shall, however, be more precise when we describe the core language used by the compiler.

A reasonable high-level programming model for parallel computers is one based on *bulk operations*, where we program in terms of transformations on entire collections of data. Suppose we wish to increment every element in a vector by 2. The functional programming tradition comes with an established vocabulary of such bulk operations which we can use as inspiration. For this task, we use the **`map`** construct, which takes a function $\alpha \rightarrow \beta$ and an array of values of type α, and produces a collection of values of type β:

```
map (\x -> x + 2) xs
```

The above is a valid expression in the Futhark programming language. We use the notation `(\x -> ...)`, taken from Haskell, to express an anonymous function with a parameter `x`. This expression does not specify *how* the computation is to be carried out, and it does not imply any accidental ordering constraints. Indeed, the only restriction is that this expression can only be evaluated after the expression that produces `xs`.

In this section, and those that follow, Futhark will be used to demonstrate the qualities of parallel programming models. Futhark is by no means the first parallel programming language, nor even the first parallel functional language. That title likely belongs to the venerable APL, which was first described in 1962 [Ive62]. Futhark is not even the first parallel language in a λ-calculus style, as it is predated by NESL [Ble96] by some twenty years. Indeed, as we shall see, Futhark is not even the most expressive such language. In fact, Futhark has more similarities than differences from other parallel functional languages. The main contributions of this thesis are the *implementation techniques* that have been developed for Futhark—its efficient mapping to GPU hardware—as well as those bits of its language design that enable said implementation. It seems likely that the implementation techniques could be applied to other functional languages of roughly the same design. For more on why I chose to construct a new programming language rather than use an existing design, see Section 2.5.

Let us return to the **`map`** expression:

```
map (\x -> x + 2) xs
```

The style of parallelism used by Futhark is termed *explicit data parallelism*. It is *explicit* because the user is required to use special constructs (here, **`map`**) to inform the compiler of where the parallelism is located, and it is *data parallel* because the same operation is applied to different pieces of data. In contrast, thread-based programming is based on *task parallelism*, where different threads perform different operations on different pieces of data (or a shared piece of data, if one enjoys de-

bugging race conditions). Task parallelism does not necessarily imply low-level and racy code with manual synchronisation and message passing, but can be given safe structure through, for example, futures or fork/join patterns.

One important property of functional data parallelism is that the semantics are sequential. The program can be understood entirely as a serial sequence of expressions evaluating to values, with parallel execution (if any) not affecting the result in any way. This decoupling of semantics and operations is typical of functional languages, and is key to enabling aggressive automatic transformation by a compiler. It is also easier for programmers to reason sequentially than in parallel. It is still important to the programmer, however, that an operation such as **`map`** is operationally parallel, which can be described through parallel cost models. One such cost model was described and subsequently proven practically feasible for the data parallel language NESL [BG96]. While a formal cost model for Futhark is outside the scope of this thesis, the similarity of Futhark to NESL suggests that a similar approach would be viable. Instead, we use the intuitive notion that certain constructs (such as **`map`**) may be executed in parallel. However, it is not always efficient to translate all *potential* parallelism into *realised parallelism* on a concrete machine (see Section 2.2.2).

2.2.1 Choice of Parallel Combinators

A persistent question when designing a programming language is which constructs to build in, and which to derive from more primitive forms. Functional languages usually prefer just a few powerful foundational constructs, on which the rest of the language can be built. In principle, **`map`** can be used to express (almost) all other data-parallel constructs. For example, we can write a summation as follows, where we assume for simplicity that the size of the input array is a power of two:

```
let sum (xs: []i32): i32 =
  let ys' =
    loop ys=xs while length ys > 1 do
      let n = length ys / 2
      in map (+) (zip ys[0:n] ys[n:2*n])
  in ys'[0]
```

An explanation of the syntax is necessary. In Futhark, both functions and local bindings are defined using the **`let`** keyword. These functions both take a single parameter, `xs` of type `[]i32`. We write the type of arrays containing elements of type `t` as `[]t`. The **`loop`** construct is specialised syntax for expressing sequential loops. Here, `ys` is the *variant parameter*, which is initialised with the value `xs`, and receives a new value after every iteration of the loop body. The array slicing syntax `ys[n:2*n]` takes elements starting at index `n` and up to (but exclusive) `2*n`.

The summation proceeds by repeatedly splitting the array in the middle, adding

each half to the other until only a single element is left. We can characterise the performance of `sum` with with a work/depth parallel cost model [BG95]. If we suppose that **map** runs in work $O(n)$ and depth $O(1)$, the function `sum` runs in work $O(n)$ and depth $O(\log n)$. This is asymptotically optimal for physically reasonable machine models. However, if executed straightforwardly on a GPU, this function will be far from reaching peak potential performance. One reason is that it is very expensive to create the intermediate `ys` arrays, compared to the time taken to add the numbers. While a "sufficiently smart" compiler may be able to rewrite the program to avoid the overhead, this is contrary to the philosophy behind Futhark: although we do not mind aggressive optimisation, the transformations performed should arise naturally from the constructs used by the programmer, and not depend on fragile analyses.

As a consequence, we provide several parallel constructs in Futhark that, while expressible in an asymptotically optimal form using **map** and sequential loops, can be mapped by the compiler to far superior low-level code. We still wish to keep the number of constructs small, because each requires significant effort to implement and integrate in the compiler, particularly with respect to optimisations such as fusion (Chapter 7). The main constructs we have chosen to include are the following, which closely resemble higher-order functions found in most functional languages:

- **map** : $(\alpha \to \beta) \to$ `[]`$\alpha \to$ `[]`β
  map `f xs ≡ [f xs[0], f xs[1], ..., f xs[n-1]]`
  Applies the function `f` to every element of the array `xs`, producing a new array.

- **reduce** : $(\alpha \to \alpha \to \alpha) \to \alpha \to$ `[]`$\alpha \to \alpha$
  reduce `f v xs ≡ f xs[0] (f ... xs[n-1])`
  Reduces the array `xs` with the function `f`. The function must take two arguments and be associative, with `v` as the neutral element. If `xs` is empty, `v` is produced. This is similar to the `fold` function found in functional languages, but the extra requirements permit parallel execution.

- **scan** : $(\alpha \to \alpha \to \alpha) \to \alpha \to$ `[]`$\alpha \to$ `[]`α
  scan `f v xs ≡ [`**reduce** `f v xs[0:1],`
  reduce `f v xs[0:2],`
  `...`
  reduce `f v xs[0:n]]`
  Computes an *inclusive scan* (sometimes called *generalised prefix sum*) of the array `xs`. As with reduction, the function must be associative and have `v` as its neutral element. The resulting array has the same size as `xs`.

- **filter** : $(\alpha \to$ `bool`$) \to$ `[]`$\alpha \to$ `[]`α
  filter `f xs ≡ [x | x <- xs, f x]`
  Produces an array consisting of those elements in `xs` for which the function `f` returns **true**.

Together, constructs such as these are called *second-order array combinators* (or *SOAC*s). Futhark does not presently permit the programmer to write their own higher-order functions, so new ones cannot be defined (this can be worked around via higher-order modules; see Section 2.5). Futhark also contains two more exotic SOACs, **`stream_red`** and **`stream_map`**, which will be discussed in Section 2.5.2.

Requiring associativity and a neutral element for the functional arguments to **`reduce`** and **`scan`** is what enables a parallel implementation. For reductions, performance can be improved further if the operator is also commutative (see Section 4.5). For simple functions, the Futhark compiler can detect commutativity, but otherwise the programmer can use a special **`reduce_comm`** SOAC that behaves exactly like **`reduce`**, but carries the promise that the function is commutative. There is no such variant for **`scan`**, because scans do not benefit from commutative operators.

In all cases, it is the programmer's responsibility to ensure that the functions have the required properties - the compiler will not check. Indeed, checking such semantic (as opposed to syntactic) properties in general is undecidable, as shown by Rice's Theorem [Ric53]. If an invalid function is provided (such as subtraction, which is not associative), the program may produce nondeterministic results.

In some parallel languages or libraries, reductions and scans are restricted to a small collection of standard operators, such as addition or maximum. This makes the language safer, but we have found many examples of problems that can be solved efficiently with nonstandard reductions. For example, consider the problem of finding the index of the largest element in an array of floating-point numbers. Because of its frequent use, this is a primitive operation in many parallel libraries and languages. In Futhark, we can express it as:

```
let index_of_max [n] (xs: [n]f32): i32 =
  let (_, i) =
    reduce_comm
      (\(x,xi) (y,yi) ->
        if      xi <  0 then (y, yi)
        else if yi <  0 then (x, xi)
        else if  x <  y then (y, yi)
        else if  y <  x then (x, xi)
        else if xi < yi then (y, yi)
        else                 (x, xi))
      (0.0, -1)
      (zip xs [0...n-1])
  in i
```

First, another note on syntax. The *size parameter* `[n]` indicates that the function is polymorphic in some size `n`. This parameter is in scope as a variable of type `i32` in the remaining parameters and body of the function. We use this to indicate that the

array parameter `xs` has `n` elements, and to construct an array of the integers from 0 to `n-1` in the expression `[0...n-1]`. We will return to size parameters in Section 2.5.

The `index_of_max` function pairs each element in `xs` with its index, then performing a reduction over the resulting array of pairs. As a result, the two operands accepted by the reduction function themselves consist of two values. We consider the neutral element to be any pair where the index is negative. To make the operator commutative, we use the indices as a tie-breaker in cases where multiple elements of `xs` have the same value, and pick the element with the greater index. We use **`reduce_comm`** instead of plain **`reduce`** to inform the compiler that the operator is indeed commutative.

2.2.2 Efficient Sequentialisation

In the literature, a *parallelising compiler* is a compiler that takes as input a program written in some sequential language, typically C or Fortran, and attempts to automatically deduce (sometimes with the help of programmer-given annotations) which loops are parallel, and how best to exploit the available parallelism. A large variety of techniques exist, ranging from sophisticated static approaches based on loop index analysis [Pou+11], to speculative execution that assumes all loops are parallel, and dynamically falls back to sequential execution of the assumption fails at runtime [OM08]. The Futhark compiler is *not* such a parallelising compiler. Instead, we assume that the programmer has already made all parallelism explicit via constructs such as **`map`** (and others we will cover). In this style of programming, it is likely that the program contains *excess parallelism*, that is, more parallelism than the machine needs. As almost all parallelism comes with a cost in terms of overhead, one of the main challenges of the Futhark compilers is to determine how much of this parallelism to actually take advantage of, and how much to turn into low-overhead sequential code via *efficient sequentialisation*. It is therefore more correct to say that the Futhark compiler is a *sequentialising compiler*.

For example, let us consider the **`reduce`** construct, which is used for transforming an array of elements of type α into a single element of type α:

```
reduce (\x y -> x + y) 0 xs
```

We require that the functional argument is an associative function, and that the second element is a neutral element for that function. The conventional way to illustrate a parallel reduction is via a tree, as on Figure 1. To reduce an n-element array $[x_1, \ldots, x_n]$ using operator $\oplus$, we launch $n/2$ threads, with thread i computing $x_{2i} \oplus x_{2i+1}$. The initial n elements are thus reduced to $n/2$ elements. The process is repeated until just a single value is left–the final result of the reduction. We perform $O(\log(n))$ partial reductions, each of which is perfectly parallel, resulting in a work depth of $O(\log(n))$.

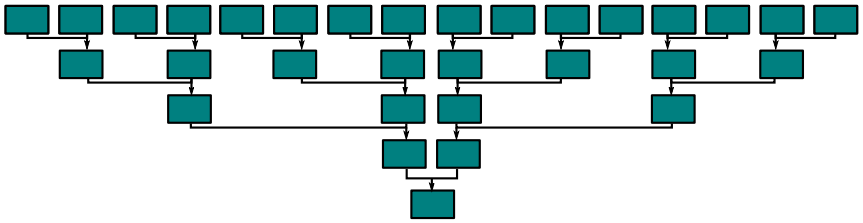


Figure 1: Summation of sixteen elements as a tree reduction.

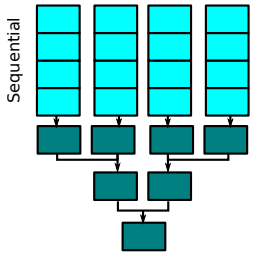


Figure 2: Summation of sixteen elements by four threads each processing four elements sequentially, then perfoming a tree reduction to combine the four per-thread results.

Tree reduction is optimal on an idealised perfectly parallel machine, but on real hardware, such as GPUs, it is inefficient. The inefficiency is caused by exposing more parallelism than needed to fully exploit the hardware. The excess parallelism means we pay an unnecessary overhead due to communication cost between threads. For a summation, the overhead of communicating intermediate results between processors significantly dominates the cost of a single addition. Efficient parallel execution relies on exposing as much parallelism as is needed to saturate the machine, but no more.

The optimal amount of parallelism depends on the hardware and exact form of the reduction, but suppose that parallel k threads are sufficient. Then, instead of spawning a number of threads dependent on the input size n, we always spawn k threads. Each thread sequentially reduces a chunk of the input consisting of $\frac{n}{k}$ elements, producing one intermediate result per thread. We then launch a second reduction over all these intermediate results. This second reduction can also be done in parallel, or can be sequential if k is sufficiently small. This approach is shown on Figure 2. A rough performance comparison between sample implementations of the two approaches to reduction is shown on Figure 3. On a sequential machine, we can simply set $k = 1$, and not exploit any parallelism at all. Efficient sequentialisation is particularly important (and also more difficult) when it comes to handling nested parallelism, as it enables locality-of-reference optimisations as we shall see in Section 2.4.

Efficient sequentialisation is not a single implementation technique, but a general implementation philosophy, which gives rise to various techniques and design choices. It is a principle that I shall often return to during this thesis, as it has proven

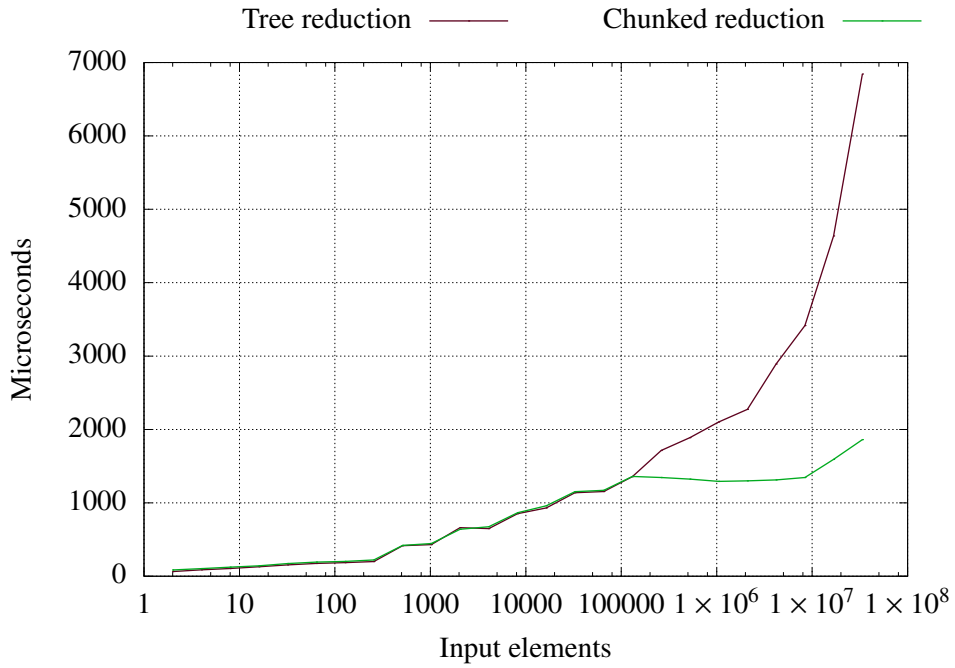


Figure 3: The runtime of a summation implemented as either a tree reduction or a chunked reduction, and executed on an NVIDIA K40 GPU. The runtime of chunked reduction follows that of tree reduction in the degenerate case where we have insufficient input to saturate the machine, after which chunked reduction performs much better. Indeed, until we reach approximately ten million input elements, the addition of extra work has no effect on runtime, as it is covered by latency hiding.

critical for executing Futhark efficiently on real hardware. In essence, efficient sequentialisation is used to bridge the impedance mismatch between the "perfectly parallel" abstract machine assumed by Futhark, and real machines that all have only limited parallelism. As we shall see, using efficient sequentialisation to move from perfect parallelism to limited parallelism is much easier, than the the efforts parallelising compilers go to when converting sequential code to parallel code.

2.3 Fusion for Modular Programming

One of the most important goals for most programming languages is to support modular and abstract programming. A modular program is composed of nominally independent subcomponents, which are composed to form a full program. The simplest feature that supports modular programming is perhaps the *procedure*. Almost all programming languages support the subdivision of a program into procedures, although most languages also support higher-level abstractions. The most important property of a procedure is that it can be understood in terms of its specification, rather than its implementation. This simplifies reasoning by abstracting away irrelevant details.

As a purely functional language, procedures in Futhark are called *functions*. To support a programming style based on the writing of small, reusable components, it is important that there is little run-time overhead to the use of functions. Function inlining is a well established technique to remove the overhead of function calls, although at the cost of an increase in code size. Another useful property of inlining is that it enables further optimisation. When an opaque function call is replaced by the function body, further simplification may be possible. While widespread techniques such as copy propagation, constant folding, and dead code removal remain useful in a data-parallel language, such as Futhark, other, more sophisticated, transformations are also important. One such transformation is *loop fusion*, which combines several loops into one. Chapter 7 discusses in more detail the technique by which fusion is implemented in the Futhark compiler. The remainder of this section discusses the intuition and motivation behind producer-consumer loop fusion[6], as well as showing how fusion is significantly easier in the functional setting than for imperative languages.

Let us consider two Futhark functions on arrays:

```
let arr_abs (xs: []i32) = map i32.abs xs

let arr_incr (xs: []i32) = map (+1) xs
```

The function `arr_abs` applies the function `i32.abs` (absolute value of a 32-bit integer) to every element of the input array. The function `arr_incr` increases every element of the input by `1`. We use a shorthand for the functional argument: `(+1)` is equivalent to `\x -> x + 1`, similarly to the *operator sections* of Haskell.

Consider now the following expression:

```
let ys = arr_abs xs
in arr_incr ys
```

If we inline `arr_abs` and `arr_incr` we obtain:

```
let ys = map i32.abs xs
in map (+1) ys
```

If we suppose a straightforward execution, the first **map** will read each element of `xs` from memory, compute its absolute value, then write the results back to memory as the array `ys`. The second **map** will then read back the elements of the `ys` array, perform the `(+1)` operation, and place the result somewhere else in memory. If `xs` has n elements, the result is a total of $4n$ memory operations. Given that memory access is often the bottleneck in current computer systems, this is wasteful. Instead, we should read each element of the array `xs`, apply the combined function

[6] Often called *vertical fusion*.

`(\x -> i32.abs x + 1)`, then write the final result, for a total of $2n$ memory operations. We could write such a **`map`** manually, but we would lose modularity, as the program is no longer structured as a composition of re-usable functions. For such simple functions as are used in this example, the loss is not great, but the issue remains for more complicated functions.

The compiler employs producer-consumer *loop fusion* to combine the two **`map`** operations into one. The validity of fusion is in this case justified by the algebraic rule

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

This permits the Futhark compiler to automatically combine the two **`map`**s and produce the following program:

```
map (\x -> let y = i32.abs x in y + 1) xs
```

Fusion is *the* core implementation technique that permits code to be written as a composition of simple parallel operations, without having to actually manifest most of the intermediate results. It is worth noting that the fusion algorithm used by the Futhark compiler respects the asymptotic behaviour of programs. Thus, a program that is fully fused will only be a constant amount faster than one that is not fused at all. This is in fact a *feature*, as it means the tractability of a program does not depend on a compiler optimisation. Chapter 7 goes into more detail.

While the Futhark programming language does not correspond directly to any specific calculus, it is heavily based on the array combinator calculus discussed in Chapter 3. This calculus serves as inspiration and justification for rewrite rules that are exploited to transform user-written programs into forms that are more efficient. As we shall see, the usual array combinators (**`map`**, **`reduce`**, **`scan`**, etc) are not adequate for fusion purposes. Using these, we cannot even perform the classic case of **`map`**-**`reduce`** fusion, as we would invariably break either the type rules for the **`reduce`** operator, or the requirement of associativity. In Section 7.1 we introduce a new set of internal combinators that are used to perform aggressive fusion.

2.3.1 How Combinators Aid Fusion

While loop fusion is not an unknown technique in compilers for imperative languages [KM93], it is significantly more complicated to implement. One major problem is that imperative languages do not have **`map`** as a fundamental construct. Hence, index analysis is first needed to determine that some loop in fact encodes a **`map`** operation, as the following imperative pseudocode demonstrates:

```
for i < n:
  ys[i] <- f(xs[i])
for i < n:
  zs[i] <- g(ys[i])
```

While index analysis can easily become undecidable, it is feasible for simple cases, such as this one. A bigger problem is that there is no guarantee that the loops can be executed in parallel. For example, the functions `f` and `g` may have arbitrary side effects, which means that they must be executed in order. Many functions written in an imperative language, even those that are externally pure, use side-effects internally, for example for storage management or to maintain accumulator variables in loops. It can be difficult for a compiler to automatically determine that a function is indeed pure. A solution is to have the programmer manually add a purity annotation to the function, which is then trusted by the compiler. There are two problems with this technique: first, the programmer may be wrong, which may result in unpredictably wrong code, depending on how the optimiser exploits the information. Second, optimising compilers are notoriously bad at providing feedback about when insufficient information inhibits an optimisation, and how the programmer can rectify the problem. A performance-conscious programmer may end up liberally sprinkling purity annotations on most of their functions in the hope of helping the optimiser, thus exacerbating the first problem.

Even if we can somehow determine `f` and `g` to be pure, the in-place assignment to `y` may have an effect if `xs` and `ys` are aliases of each other (i.e. overlap in memory). Alias analysis is one of the great challenges for compiler optimisation in imperative languages [HG92]—indeed, aliasing guarantees are one of the performance benefits Fortran has over languages such as C. Modern languages tend to support annotations by which the programmer can indicate that some array has no aliases in scope[7]. These annotations, while useful, have the same issues as the purity annotations discussed above.

In a purely functional language, we avoid these issues by construction. This allows the compiler (and the compiler writer!) to focus on *exploiting* parallel properties, rather than *proving* them.

2.4 Nested Parallelism

In a parallel expression **`map`** `f xs`, the function `f` can in principle be anything. In particular, `f` can contain more parallelism. When one parallel construct is nested inside of another, we call it *nested parallelism*.

The need for nested parallelism arises naturally out of our desire to support modular programming. We should be able to map any function `f`, even if `f` is parallel itself.

[7] The `restrict` keyword in C99.

Furthermore, the parallelism inside of `f` should also be utilised—it's not enough to exploit only the outermost level of parallelism, as that may not be enough to saturate the hardware. Unfortunately, it turns out that nested parallelism is difficult to implement efficiently in its full generality. The reason is that it is hard to map arbitrary nested parallelism to current parallel hardware, which supports only a fixed level of parallelism efficiently (and typically with harsh restrictions on the size of each level beyond the first; see Chapter 4). This is an example of an impedance mismatch between free-form nested parallel program and the real hardware we have available to us. Fortunately, there are ways to bridge the divide that follow straightforwardly from the construction of data-parallel functional programs in general, and Futhark programs in particular.

Guy Blelloch's seminal work on NESL demonstrated how to handle arbitrary nested parallelism via *full flattening* [Ble+94]. The flattening algorithm transforms arbitrary nested data parallelism into flat data parallelism, which can be easily mapped on to most machines. The key technique is *vectorisation*, by which each function f is lifted to a vectorised version $\hat{f}$, that applies to *segments* of some larger array. While flattening is useful for its universal applicability, it has three main problems:

1. *All* parallelism is exploited, even that which is expensive to exploit (perhaps hidden behind branches) and not necessary to take full advantage of the hardware.

2. The vectorisation transformation forces all sequential loops to the outermost level, thus preventing low-overhead sequential loops inside threads. This is particularly harmful for programs that are "almost flat", such as a **`map`** whose function simply performs a sequential loop. Flattening would transform this into a sequential loop that contains a **`map`**, thus forcing an array of intermediate results to be written after every **`map`**. In contrast, the original loop may have been able to run using just registers for storage.

3. The structure of the original program is heavily modified, destroying much information and rendering optimisations based on access pattern information (such as loop tiling) infeasible.

Work is still progressing on adapting and improving the flattening transformation. For example, [Kel+12] shows how to avoid vectorisation in places where it produces only overhead with little gain in parallelism, particularly addressing problem (2) above.

Flattening remains the only technique to have demonstrated universal applicability, and is thus useful as a "last resort" for awkward programs that admit no other solution. However, many interesting programs only exhibit limited nested parallelism. Specifically, they exhibit only *regular* nested parallelism, which is significantly easier to map to hardware.

Nested parallelism is regular if the amount of parallelism in its inner parallel loops is invariant to its outer parallel loops, and otherwise *irregular*. For example, the following Futhark expression contains irregular parallelism:

```
map (\i -> reduce (+) 0 [1...i]) [1...n]
```

While this one does not:

```
map (\i -> reduce (+) 0 (map (+i) [1...n])) [1...n]
```

In the former program, the inner parallel **reduce** operates on an array containing `i` elements, where `i` is bound by the function of the **map**. In the latter case, the **reduce** operates on an array of size `n`, where `n` is bound outside the expression. The parameter `i` is still used, but it does not contribute to the size of any array, only to their values. While the Futhark language does support irregular nested parallelism, as demonstrated above, the current implementation is not able to exploit it. Should it prove necessary, the Futhark compiler could be modified to incorporate the flattening algorithm as well, but for this thesis, we have focused on developing implementation techniques that are less general, but produce faster code.

The limitation to regular nested parallelism is not as onerous as it may seem. First, many interesting problems are naturally regular (see Chapter 10 for examples). Second, we can always *manually* apply the flattening algorithm to our program to the degree necessary to remove irregular nested parallelism. Of course, this may require manual inlining, thus breaking modularity. Third, many irregular programs can be modified in an ad-hoc fashion to become regular. For example, the irregular program shown above can be rewritten to

```
map (\i -> reduce (+) 0
            (map (\x -> if x > n then 0 else x)
                 [1...n]))
    [1...n]
```

But this method is not mechanical—a unique approach is required based on the algorithm in question, in contrast to flattening, which is general.

The largest problem with the restriction to only regular parallelism is that it inhibits modularity. Consider a function that computes the sum of the first n positive integers for some n:

```
let sum_nats (n: i32): i32 =
  reduce (+) 0 [1...n]
```

The type of this function is merely `i32 -> i32`, and so we should be able to map it over an array of type `i32`:

```
map sum_nats [1...100]
```

However, this gives rise to irregular parallelism, because the parallelism of the **map** inside the definition of `sum_nats` depends on the integer argument to `sum_nats`. This is not a problem that can be detected merely by the type of `sum_nats`. It remains future work to investigate language mechanisms that allows programmers to reason about the regularity of parallelism in a modular fashion.

The restriction to regular parallelism is an artifact of the current Futhark implementation, not Futhark as a programming language. Therefore, if necessary, an implementation could be constructed that avoids this issue by using full flattening. However, the performance advantages of regular nested parallelism still motivates a language mechanism for modular reasoning.

2.5 Why a New Language?

Most of the preceding discussion involves concepts and constructs that are common to many functional languages. Indeed, it has often been claimed that pure functional programming makes parallel execution straightforward. Why, then, do we propose an entirely new language, rather than applying our techniques to existing and reasonably popular languages such as Haskell, OCaml, or F#? What does Futhark have that these languages lack?

The answer lies in the inverse question: what has been excluded from Futhark to permit efficient parallel execution? While all mentioned languages contain the essentials for data-parallel programming—bulk operations with pure functions—they also contain features and promote programming styles that complicates the task of writing a compiler capable of generating high-performance code for restricted parallel machines, such as GPUs. This is not a task that needs further complication. The following discusses common problematic traits, and how Futhark circumvents them. We do not claim that these problems cannot be solved, merely that they provide significant challenges that would have distracted from the core goal of our work, which is developing compilation techniques for programs written as compositions of bulk parallel operators.

Emphasis on recursion: Most forms of recursion are an inherently sequential process, which is accordingly of less general usefulness in a parallel language. However, even for those cases where we do desire sequential loops, free-form recursive functions prove challenging to implement efficiently on a GPU.

On the GPU, threads do not possess a stack in the conventional sense. While stacks can be emulated (slowly), hardware limitations force a small, static size. Consider, for example, a GPU that requires $100,000$ parallel threads to be saturated. Should this GPU possess 8GiB of memory (a large size for the

current generation), each thread could at most have a stack size of 83KiB—and only if we use all available memory just for stacks. Recursion inside code running on a GPU is therefore a bad idea.

A language could conceivably be defined where the only form of recursion permitted is tail-recursion, which does not suffer from this problem. In the interest of expedience, Futhark entirely bans ordinary recursive functions, and provides special syntax (the `loop` construct) for the equivalent of tail recursion. In the future, this restriction could be loosened to support (mutually) tail recursive functions.

Another approach, which is in practice what is done by the flattening transformation, is to interchange the recursion outside of the parallel sections. However, this comes at a significant cost in both memory usage and expensive control flow [Kel+12].

Sum types and recursive types: Sum types by themselves are only problematic in that they necessarily imply control flow to handle different cases. While control flow is not hard to map to a GPU, branching can be expensive. Nevertheless, the use of a sum type is an explicit choice made by the programmer, presumably with an understanding of the costs involved, and so are likely to be supported by Futhark in the future.

Recursive types (such as linked lists) usually imply pointer-chasing at runtime, which fits poorly with the very regular access patterns require to obtain peak GPU performance (see Section 4.3.1). Constructing complex data types usually requires allocation, which is also generally not possible on a GPU. While it has been shown that these problems can be solved in some cases, for example by region inference in the Harland programming language [Hol+14], it is not clear that the performance of the resulting code is good. As Futhark is a language primarily focused on obtaining good performance, we have left out features that—while *possible* to map to GPU—should probably not be used in a program we wish to be fast.

Lazy evaluation: Lazy evaluation is essentially shared state plus runtime effects—two of the most difficult things to map to GPUs. As allocation is not generally possible on the GPU, it is a serious problem that evaluation of any term could potentially require allocating an arbitrary amount of space. It is telling that data-parallel extensions to Haskell, such as Accelerate [McD+13] and Data Parallel Haskell [Cha+07], are both strict, even though Haskell itself heavily emphasises laziness.

Side effects: While all functional languages emphasise programming with pure functions, most languages in the ML family tend to permit essentially unrestricted side effects. This is traditionally not a problem when only "benign effects" are

used. For example, a function could internally use mutable references for performance reasons, such as to implement memoisation. As long as the effects are not visible to callers of the function, the programmer can still perform black-box reasoning as if the function were pure.

However, a compiler does not have this freedom. Even if we are willing to trust some annotation that an function implemented with effects really behaves as an extensionally pure function, this does not extend to its definition. A compiler cannot treat a function as a black box, but may need to perform significant transformations on its definition to produce efficient code.

As a result, Futhark permits no side effects, except for nontermination and other error cases. To address some of the performance issues associated with expressing efficient sequential algorithms in a pure functional language, Futhark supports *uniqueness types*, which are used to permit in-place updates of arrays under some circumstances. These are discussed in Section 2.5.1.

First class functions: GPUs do not support function pointers (related to the absence of a stack), which immediately renders most conventional implementation techniques for first-class functions impractical. We can employ *defunctionalisation* [Rey72], where every function term in the program is identified with a unique integer, and a large branch statement is used to select the corresponding function a runtime. This is the approach taken by Harland [Hol+14], but the heavy use of branching renders it inefficient on GPUs. In particular, this approach has the unfortunate consequence that whenever the programmer adds a function, all other function calls will become slower (except those that can be statically resolved).

While fully first class functions are not viable, limited support for higher-order functions is possible. Whenever we can statically determine the form of the functional arguments to a function, we can generate a specialised version of the function at compile-time, where the concrete functional argument is embedded. This is not yet implemented in Futhark, and so user-defined higher-order functions are not supported. It is, however, possible to use the higher-order module system to imitate higher-order functions, albeit at some cost in boilerplate. An example is shown on Figure 4. As the module system is outside the scope of this thesis, we shall not delve further into the merits and demerits of this approach.

The primary motivation behind Futhark was to create a language whose efficient implementation is not hindered by complicated features. For the most part, Futhark resembles a least-common-denominator functional language. However, we have taken the opportunity to add various language features specialised for the task of high-performance parallel array programming. In particular, the fact that Futhark

```
-- A parametric module (functor) taking as argument
-- a module, and producing a module.
module map2(fm: { type c_t -- "closure" type
                  type x_t -- element type
                  val f: c_t -> x_t -> x_t }) = {
  let g [n] (c: fm.c_t) (xs: [n]fm.x_t): [n]fm.x_t =
    map (\x -> fm.f c (fm.f c x)) xs
}

module increment_twice = map2 {
  type c_t = i32
  type x_t = i32
  let f (c: i32) (x: i32) = x + c
}

module scale_twice = map2 {
  type c_t = f64
  type x_t = i32
  let f (c: f64) (x: i32) = t64 (r64 x * c)
}

let foo = increment_twice.g 2 [1...5]
-- foo == [5i32, 6i32, 7i32, 8i32, 9i32]
let bar = scale_twice.g f64.pi [1...5]
-- bar == [9i32, 18i32, 28i32, 37i32, 47i32]
```

Figure 4: Using higher-order modules to imitate higher order functions in Futhark. The `map2` parametric module produces a module containing a function `g`, which applies a provided function twice. The parameter of type `c_t` is used to emulate a closure environment.

```
let dotprod [n] (xs: [n]i32) (ys: [n]i32): i32 =
  reduce (+) 0 (map (+) (zip xs ys))
```

(a) Dot product of integer vectors in Futhark.

```
let matmul [n][m][p]
    (xss: [n][m]i32) (yss: [m][p]i32): [n][p]i32 =
  map (\xs -> map (dotprod xs) (transpose yss)) xss
```

(b) Multiplication of integer matrices in Futhark.

Figure 5: Two examples of using size parameters to encode size constraints on Futhark functions. Presently the constraints are checked at run-time, not compile-time, and thus serve more as documentation than a safety mechanism.

is not intended for constructing advanced user-defined data types, but instead expects programmers to express their program in terms of arrays, allows us to add specialised constructs and notation to aid in array programming. We have already seen an example in the form of size parameters, which allows the programmer to express size constraints on functions parameters. For example, the dot product function in Figure 5a requires that the two input arrays have the same length, while the matrix multiplication in Figure 5b specifies the usual size constraints on matrix multiplication.

Apart from the usual parallel combinators, Futhark also supports two somewhat more exotic features, both of which are introduced for performance reasons. These are discussed below.

2.5.1 In-Place Updates

While Futhark is a data-parallel language, and expects programs to exhibit large amounts of parallelism, it also supports the implementation of limited imperative algorithms in an efficient manner. Specifically, Futhark supports *in-place updates*, which allows one to create an array that is semantically a copy of another array, but with new values at specified positions. This can be done in a purely functional manner by simply copying the array, but this would cause the asymptotic cost of the update to be proportional to the full size of the array, not just the part that we are updating. With an in-place update, we pay only for the part of the array that we are updating. This section describes how Futhark supports in-place updates without violating the functional purity that we depend on for safe parallel programming.

Before further discussion, we must justify why in-place updates are needed. Our motivation is twofold. First, it is not uncommon for a parallel program to consist

of a number of parallel loops surrounding an inner sequential code. The simplest instance of this pattern is mapping a function `f`, that internally performs side effects but is externally pure, over every element of an array. It is just as important that the inner sequential code is efficient, as it is that we execute the outer loop in parallel. Both influence the final program performance.

Our second piece of motivation is the notion of efficient sequentialisation. In Section 2.5.2 we shall see language constructs that permit the programmer to describe a computation comprising both an efficient sequential part, as well as a description of how to combine sequentially produced results in parallel. The utility of such language constructs hinges entirely on the ability to actually express said efficient sequential code. In-place updates are key to this ability.

The main language construct that permits array update is quite simple:

```
a with [i] <- v
```

This expression semantically produces an array that is identical to `a`, except with the element at position `i` replaced by `v`. The compiler then verifies that no alias of `a` (including `a` itself) is used on any execution path following the in-place update. If the compiler had to perform a full copy of `a`, the asymptotic cost would be $O(n)$ (where n is the size of `a`). Using an in-place update, the cost is $O(1)$ (assuming that `v` has constant size).

Uniqueness Types

The safety of in-place updates is supported in Futhark through a type-system feature called *uniqueness types*. This system is similar to, but simpler than, the system found in Clean [BS93; BS96], where the primary motivation is modeling IO. Our use is reminiscent of the ownership types of Rust [Hoa13]. Alongside a relatively simple conservative and intra-procedural aliasing analysis in the type checker, this approach is sufficient to determine at compile time whether an in-place modification is safe, and signal an error otherwise.

This section describes uniqueness types in intuitive terms. A more precise formalisation on the Futhark core language can be found in Section 5.3. An important concept used in our uniqueness type system is *aliasing*. Each array-typed variable is associated with a set of other variables in scope, which it may alias. Operationally, aliasing models the possibility that two arrays overlap are stored at overlapping memory locations. A freshly created array (as by **`map`** or an array literal) has no aliases. Some language constructs, like array slicing, may produce arrays that are aliased with the source array. The result of **`if`** aliases the union of the aliases of both branches.

When an array is modified in-place, all of its aliases are marked as *consumed*, and a compile-time error occurs if they are used afterwards. This prevents the in-place modification from being observable, except for its performance effects. The larger

problem is how to ensure safety in the interprocedural case, which is where uniqueness types enter the picture. We introduce uniqueness types through an example, which shows a function definition:

```
let modify [n] (a: *[n]int) (i: int)
               (x: [n]int): *[n]int =
  a with [i] <- (a[i] + x[i])
```

A call `modify a i x` returns `a`, but where `a[i]` has been increased by `x[i]`. In the parameter declaration `(a: *[n]int)`, the asterisk (`*`) means that `modify` has been given "ownership" of the array `a`. The caller of `modify` will never reference array `a` after the call. As a consequence, `modify` can change the element at index `i` in place, without first copying the array; i.e. `modify` is free to do an in-place modification. Further, the result of `modify` is also unique—the `*` in the return type declares that the function result will not be aliased with any of the non-unique parameters (it might alias `a` but not with `x`). Finally, the call `modify a i x` is valid if neither `a` nor any variable that aliases `a` is used on any execution path following the call to `modify`.

We say that an array is *consumed* when it is the source of an in-place update or is passed as a unique parameter to a function call; for instance, `a` is consumed in the expression `a with [i] <- x`. Past the consumption point, neither `a` nor its aliases may be used again. From an implementation perspective, this contract is what allows type checking to rely on simple intra-procedural analysis, both in the callee and in the caller, as described in the following sections.

Parallel Scatter

The in-place update construct shown above is satisfactory for sequential programming. However, the uniqueness type system also lets us define an efficient construct for *parallel scatter*. A scatter operation, whose nomenclature is taken from vector processing, takes a destination array `a`, an array of indices `js`, and an array of values `vs`, and writes each value to the position in the array given by the corresponding index. The operation can be explained in imperative pseudocode as follows:

```
for i in 0...n-1:
  a[js[i]] = vs[i]
```

In Futhark, the **`scatter`** construct is used as follows:

```
scatter a js vs
```

Semantically, **`scatter`** returns a new array. We treat the array `a` as consumed, which operationally permits the compiler to perform the operation in-place, thus making the cost of **`scatter`** proportional to the size of `vs`, not `a`.

The **scatter** construct is fully parallel, and therefore it is unspecified what happens if we try to write distinct values to the same index. Other parallel languages, for example Accelerate [McD+13], contain a variation of **scatter** that requires the specification of an associative and commutative combination function, which is used to combine duplicate writes. While this carries nontrivial extra run-time overhead, it makes the construct safer and more flexible, and will likely be added to Futhark in the future.

The **scatter** construct is primarily used as an "escape hatch" for encoding irregular problems that are not a good fit for any of the conventional SOACs. An example is discussed in Section 10.2.

In vector programming, **scatter** is often paired with a matching "gather" construct. There is no need for this in Futhark, as a gather is easily expressible as a **map** over an index space, as follows:

```
map (\i -> a[i]) is
```

2.5.2 Streaming SOACs

Most of the array combinators supported by Futhark are familiar to users of existing functional languages. However, we found it useful to add two novel SOACs that directly address our focus on efficient sequentialisation. This section introduces the *streaming SOACs*, **stream_red** and **stream_map**, by demonstrating their application to k-means clustering and generation of Sobol numbers. The idea behind the streaming SOACs is to allow the programmer to provide an efficient sequential algorithm which is applied in parallel to separate *chunks* of the input, with the per-chunk results combined via a programmer-provided function. This is useful when the hand-written sequential algorithm is more efficient than simply executing the parallel algorithm sequentially.

k-means clustering

In multidimensional cluster analysis, one widely used algorithm is k-means analysis. The goal is to assign n points in a d-dimensional space to one of k clusters, with each point belonging to the cluster with the closest centre. The centre of a cluster is the mean of all points belonging to the cluster. Often, k is small (maybe 5), while d can be larger (27 in one of our data sets), and n usually much larger (hundreds of thousands). The algorithm is typically implemented by initially assigning points randomly to clusters, then recomputing cluster means and re-assigning points until a fixed point is reached. In this section, we explain how one part of the algorithm—the computation of cluster centres—can be efficiently expressed using **stream_red**, and its performance advantage over a traditional **map**-**reduce**-

```
1  let add_centroids [d] (x: [d]f32, y: [d]f32)
2                        : *[d]f32 =
3    map (+) (zip x y)
4
5  let cluster_sums_seq [k] [n] [d]
6                       (counts: [k]i32)
7                       (points: [n]([d]f32,i32))
8                       : [k][d]f32 =
9    loop acc = replicate k (replicate d 0.0)
10     for (p,c) in points do
11       let p' = map (/f32(counts[c])) p
12       in acc with [c] <- add_centroids acc[c] p'
```

Figure 6: Sequential calculation of means.

formulation. The problem is taken from the k-means benchmark from the Rodinia benchmark suite [Che+09], which is discussed further in Chapter 10.

As a first approximation—and also as further introduction to Futhark—we will discuss a (mostly) sequential solution of the problem. This will also be a useful building block in our final solution. The code is shown in Figure 6. On lines 1–3 we define a function for adding d-dimensional points, which we represent as vectors of floating-point values. Element-wise addition is done by passing both vectors to **map** with the addition function as the operator. We mark the return value as unique, meaning that it does not alias either of the input parameters. The need for this will be apparent shortly.

On lines 9–12, we define the main loop. This loop proceeds by maintaining an accumulator `acc` that contains the cluster centres as they have been computed thus far. The initial value is constructed via the **replicate** array constructor, whose first argument is the number of times to replicate the second argument. In this case, two nested **replicate**s are used to construct a two-dimensional array. We then iterate across all points, and for each point we add its contribution to the corresponding cluster.

Line 12, which updates `acc`, is particularly interesting, as it performs an in-place update. Semantically, we construct a new array that is a copy of `acc`, except that the value at the given index has been replaced by the result of the call to `add_centroids`, and bind this array to a new variable named `acc` (shadowing the old). However, we are guaranteed that no copying takes place, and that the cost is proportional only to the size of the value we are writing (an array of size d), rather than the array we are updating (of size $k \times d$). It is important that we know that the result of `add_centroids` cannot alias its arguments, `acc[c]` or `p'`, and this is exactly

the property that is ensured by marking the return value of `add_centroids` as unique. If the return value of `add_centroids` *did* alias the `acc` array, then the in-place update into `acc` might not be safe, as we would be simultaneously reading and writing from the same array.

The `cluster_sums_seq` function forms the sequential implementation of cluster summation. While a small amount of **`map`**-parallelism is present in the function `add_centroids`, the main bulk of the work is the outer loop of n iterations. Even though the implementation performs $O(n \cdot d)$ work, which is efficient, the sequential depth is n, which is not satisfactory.

A fully parallel implementation is shown in Figure 7. The algorithm computes, for each point, an *increment matrix*, which is an array of type `[k][d]f32` containing all zeroes, except at the row corresponding to the cluster for the point in question. This computation is done on lines 5–9. In-place updates are used only in a trivial form. The result is an array of matrices, which we then sum on lines 11–13 to get the final result. The **`rearrange`** construct is a generalisation of transposition, and produces a view of the given array where the dimensions have been reordered by the given permutation. For example, if

```
b = rearrange (1,2,0) a,
```

then

```
b[i0,i1,i2] == a[i1,i2,i0].
```

In the program, the **`rearrange`** construct is used to bring the dimension we wish to reduce across innermost, which allows us to avoid a reduction where the operator operates on large arrays.[8]

This implementation exploits all degrees of parallelism in the problem. Unfortunately, it is not work efficient: constructing the increment matrix involves $O(n \cdot k \cdot d)$ work, as does its eventual reduction, while the sequential version requires only $O(n \cdot d)$ work. If executed on a system capable of exploiting all available parallelism, this might be an acceptable tradeoff, but real hardware is limited in the amount of parallelism it can take advantage of. We need a language construct that can expose enough parallelism to take full advantage of the machine, but that will run efficient sequential code within each thread. The **`stream_red`** SOAC provides just this functionality.

The **`stream_red`** construct builds on the property that any fold with an associative operator $\odot$ can be rewritten as a fold over chunks of the input, followed by a fold over the per-chunk results:

$$\texttt{fold}\ \odot\ xs = \texttt{fold}\ \odot\ (\texttt{map}\ (\texttt{fold}\ \odot)\ (\texttt{chunk}\ xs))$$

By selecting the number of chunks such that we obtain enough parallelism by the outer `map`, we can implement the innermost fold as a work-efficient sequential function. In Futhark, we let the programmer specify this *chunk function* directly.

[8] Chapter 8 shows that the Futhark compiler can do this automatically, but here we do it by hand for clarity.

```
1  let cluster_sums_par [k] [n] [d]
2                       (counts: [k]i32)
3                       (points: [n]([d]f32,i32))
4                       : *[k][d]f32 =
5    let increments: [n][k][d]i32 =
6          map (\(p, c) ->
7            let a = replicate k (replicate d 0.0)
8            in a with [c] <- map (/f32 counts[c]) p)
9               points
10   let increments': [k][d][n]i32 =
11     rearrange (1,2,0) increments
12   in map (\x -> map (\y -> reduce (+) 0.0 y) x)
13          increments'
```

Figure 7: Parallel calculation of means (types of `increments` and `increments'` annotated for clarity).

```
1  let cluster_sums_stream [k] [n] [d]
2                          (counts: [k]i32)
3                          (points: [n][d]f32)
4                          (membership: [n]i32)
5                          : [k][d]f32 =
6    stream_red
7      (\(x: [k][d]f32) (y: [k][d]f32) ->
8         map add_centroids (zip x y))
9      (cluster_sums_seq counts)
10     (zip points membership)
```

Figure 8: Chunked parallel calculation of means.

As shown in Figure 8, **`stream_red`** is given the associative reduction operator (lines 7–8), together with the *chunk function* (line 9). Because we wish to process two arrays—both `points` and `membership`—we use **`zip`** to combine the two arrays into one array of pairs (line 10). In this case, the reduction operator is matrix addition, and the chunk function applies the sequential `cluster_sums_seq` function (partially applied to `counts`) previously shown in Figure 6.

It is the programmer's responsibility to ensure that the provided reduction function is associative, and that the result of **`stream_red`** is the same no matter how the input is partitioned among chunks.

The **`stream_red`** formulation can be automatically transformed into the fully

Program	Version	Runtime	Speedup
Cluster means	Chunked (parallel)	17.6ms	×7.60
	Fully parallel	134.1ms	
	Chunked (sequential)	98.3ms	×0.92
	Fully sequential	90.7ms	
Sobol numbers	Chunked (parallel)	3.9ms	×11.13
	Fully parallel	43.4ms	
	Chunked (sequential)	129.7ms	×1.00
	Fully sequential	129.1ms	

Table 1: Speedup of chunking SOACs versus fully sequential and fully parallel implementations. For comparing sequential performance, a compiler generating single-threaded CPU code has been used and the code runs on an Intel Xeon E6-2570. For comparing parallel performance, OpenCL code is generated and executed on an NVIDIA Tesla K40 GPU. We generate $30,000,000$ Sobol numbers, and compute cluster means with $k = 5$, $n = 10,000,000$, and $d = 3$.

parallel implementation from Figure 7 by setting the chunk size to 1, or into the sequential implementation from Figure 6 by setting the chunk size to the full size of the input (followed by simplification). It thus allows the compiler or runtime system to generate code that best fits the problem and target machine. The compiler is free to exploit the nested parallelism inside both the reduction function and the fold function. In general, there is no *one size fits all*, and different parallelisation patterns are best for different data sets. Handling this issue is future work (see Section 8.3), and for now the Futhark compiler uses various hardcoded heuristics to determine how parallelisation is best done (Chapter 8). For this program, the reduction function will be fully parallelised as a segmented reduction, and the body of the chunk function will be sequentialised.

The performance of the **`stream_red`** version compared to explicitly parallel and sequential code is shown in Table 1. We see that the fully parallel version running on a GPU is in fact slower than the sequential version, because it does significantly more work. The **`stream_red`** version performs well both when compiled to parallel code, and when compiled to sequential code.

Sobol Sequences

This section introduces the **`stream_map`** SOAC. The goal is the same as previously: we wish to formulate our program in a way that gives the compiler freedom to exploit exactly as much parallelism as is profitable on the target hardware.

The problem is as follows: we wish to generate n entries from a Sobol sequence [BF88], and compute their sum. Sobol sequences are quasi-random low-discrepancy sequences frequently used in Monte-Carlo algorithms. While the summation is contrived, the need to efficiently generate Sobol sequences comes from

```
1  let gray_code (x: i32): i32 =
2    (x >> 1) ^ x
3
4  let test_bit (x: i32) (ind: i32): bool =
5    (x & (1 << ind)) == (1 << ind)
6
7  let sobol_ind [n] (dir_v: [n]i32) (x: i32): i32 =
8    let reldv_vals = map (\dv i ->
9                            if test_bit (gray_code x) i
10                           then dv else 0)
11                       dir_v [1..<n]
12   in reduce (^) 0 reldv_vals
13
14 let sobol_ind_r [n] (dir_v: [n]i32) (x: i32): f32 =
15   f32 (sobol_ind dir_v x) / (2.0 ** f32 n)
16
17 let sobol_par [n] (k: i32) (dir_v: [n]i32): f32 =
18   let sobol_nums =
19     map (sobol_ind_r dir_v) (map (+1) (iota k))
20   in reduce (+) 0.0 sobol_nums
```

Figure 9: **map**-parallel calculation of Sobol numbers.

the OptionPricing benchmark presented in Section 10.1.3. Sobol numbers can be computed by a **map**-parallel formula, or by a cheaper (recurrence) one, but which requires **scan** [And+16; Oan+12]. This property can be expressed elegantly with **stream_map**, whose function accepts an input chunk, and must produce an output chunk of the same size. Chunks are processed in parallel. For this program, the chunk function applies the **map**-parallel formula once, then applies the **scan** formula.

An implementation of the `map`-parallel formula to a sequence of k Sobol numbers is shown in Figure 9. The `[1..<n]` expression creates an array of the integers from 0 to `n-1`, and the precise Sobol sequence generated is defined by the *direction vector* `dir_v`. The independent formula works by essentially performing a **map-reduce** computation for every Sobol number, with the number of iterations being the size of the direction vector. Typically, this number is fairly small (perhaps 31) as it is limited to the number of bits in an integer.

An implementation of the recurrent formula is shown in Figure 10. Notice that for each chunk, we first apply the function `sobol_ind` to compute the first Sobol number, then apply a combination of **map** and **scan** to compute the rest of the chunk. While **map** and **scan** are parallel operators, the compiler will sequentialise them during code generation, and instead use the chunk size to control how much

```
1  let index_of_least_significant_0 (x: i32): i32 =
2    loop (i = 0) while i < 32 && ((x>>i)&1) != 0 do i + 1
3
4  let rec_m [n] (dir_v: [n]i32) (i: i32): i32 =
5    let bit = index_of_least_significant_0 i
6    in dir_v[bit]
7
8  let sobol_chunk [n]
9                  (dir_v: [n]i32)
10                 (x: i32)
11                 (chunk: i32)
12                 : [chunk]f32 =
13   let sob_beg  = sobol_ind dir_v (x+1)
14   let contrbs  = map (\i -> if i==0 then sob_beg
15                                     else rec_m dir_v (i+x))
16                      [0..<chunk]
17   let vct_ints = scan (^) 0 contrbs
18   in map (\y -> r32 y / (2.0 ** r32 n)) vct_ints
19
20 let sobol_stream [n] (k: i32) (dir_v: [n]i32): f32 =
21   let sobol_nums = stream_map
22                      (\[chunk] (xs: [chunk]i32): [chunk]f32 ->
23                         sobol_chunk dir_v xs[0] chunk)
24                      [0..<k]
25   in reduce (+) 0.0 sobol_nums
```

Figure 10: Chunked calculation of Sobol numbers.

parallelism to exploit.

Note that the two formulae are algorithmically very different. There is little chance that a compiler can derive one from the other. The performance of the **stream_map** version compared to explicitly parallel and sequential code is shown in Table 1. The significant speedup of the chunked over the fully parallel implementation is due to the inner parallelism being efficiently sequentialised through the "**stream_seq** fusion" technique discussed in Section 7.3.2. This changes the per-chunk memory footprint from O(chunksize) to $O(1)$; effectively transforming the problem from memory-bound to compute-bound.

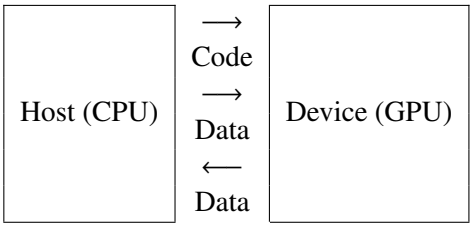


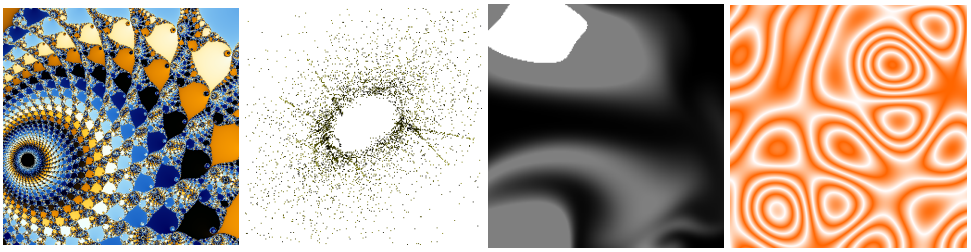
Figure 11: The run-time structure of a program using OpenCL. The "host" is typically a CPU, and is entirely in control of the "device", which might for example be a GPU, but could be anything (even the same CPU as the host).

2.6 Interoperability

Futhark is not a general-purpose programming language. As a purely functional language, interacting with the outside world is impossible. As an array-oriented language, complicated data structures are awkward or impossible. Futhark is at its core designed around trading flexibility for the ability to automatically obtain good runtime performance. As a result, writing full applications in Futhark is impossible. While the Futhark compiler can compile a Futhark program into an executable, the resulting program simply executes the Futhark `main` function, while reading input data from standard input and writing the result on standard output. This is useful for testing, but not particularly useful for an application programmer.

Futhark is intended to be used only for the computationally intensive parts of a larger application, with the main part of the application written in a general-purpose language. It is therefore important that Futhark code can be easily integrated with code written in other languages. Fortunately, the design of OpenCL [SGS10], the library used by the Futhark compiler to communicate with GPUs or other accelerators, fits this use case well.

The overall structure of OpenCL is shown on Figure 11. The CPU, which in OpenCL nomenclature is called the *host*, is in control. The host communicates with one or more *devices*, which act as coprocessors. The device can for example be a GPU, but any kind of computational resource can be presented to OpenCL via the device abstraction. The host issues commands to the devices, which may involve uploading code (kernels) or data, initiating execution of uploaded code, or retrieving data stored on the device in response to executions. The code uploaded to the device must be specified in OpenCL C, a severely restricted variant of the C kernel language, that lacks support for features such as function pointers, recursion, or allocation. The key property that we exploit for interoperability purposes is that the code running on the host is required only for managing the device, and thus need not be particularly efficient, as the vast majority of the computation takes place inside OpenCL kernels. This allows us to specify the host-level code in ways that maximise interoperability, without worrying overmuch about performance losses.



(a) Mandelbrot **(b)** *N*-body **(c)** Fluid **(d)** Crystal

Figure 12: Visualisations of four benchmark programs implemented in Futhark, and called from Python.

As a concrete example, the Futhark compiler contains a code generator backend where the host code is generated in the Python programming language, and interacts with OpenCL through the PyOpenCL library [Klö+12]. The result of compiling a Futhark program is a Python module that defines a class, the methods of which correspond to Futhark functions. These methods accept and return Python values (which may be handles to GPU data), and feel to the programmer like calling any other Python library. But behind the scenes, data and code is submitted to the GPU. This allows us to use Futhark for the heavy lifting, while making use of Python's wealth of libraries for such things as user interaction and data format serialisation. As an example, several of the benchmark programs we discuss in Chapter 10 have been furnished with interactive visualisations via the Pygame library. The Python program reads input events from the user, and the Futhark programs creates screen images (in the form of two-dimensional arrays of integers), which are then blitted to the screen via Pygame. Examples of the resulting visualisations can be seen in Figure 12

While Python is a worthwhile target language due to its popularity for scientific computing, it is not realistic to construct code generators for all languages of interest. Even a short list might include Ruby, R, C#, Java, Swift, and more. While writing a host-code backend for the Futhark compiler requires only a few thousand lines of Haskell code, we do not wish the burden of maintaining a dozen such backends. Rather, we should invest effort in improving the C backend, such that it can generate code that can be called by the foreign function interfaces (FFIs) of other languages. Due to its ubiquity, most languages make it easy to call C code. However, the Python backend still demonstrates clearly how Futhark can be useful in an application development context.

An entirely different approach is to use Futhark as a *target language* for a compiler. In fact, this was the original purpose for Futhark, but it has receded into the background as the Futhark source language grew more pleasant to use. We have, however, cursorily investigated this approach by using Futhark as the target language for an APL compiler [Hen+16], where the generated code performed comparably with hand-written Futhark.

2.7 Related Work on Automatic Parallelisation

This section discusses three simplified Fortran 77 examples from the DYFESM and BDNA benchmarks of the PERFECT club suite [Ber+88], whose automatic parallelisation requires complex compiler analysis [OR12; OR15; OR11]—for example based on (i) inter-procedural summarization of array references, (ii) extraction of runtime (sufficient) conditions for safe parallelization, and (iii) special treatment of a class of induction variables (CIV) that cannot be expressed as a closed-form formula in terms of loop indices. Such analyses are beyond the capabilities (or patience) of most programmers and commercial compilers, and it would not be necessary if application parallelism was expressed explicitly by means of bulk-parallel operators, as in a data-parallel language.

The first example, presented in Section 2.7.1, semantically corresponds to a map applied to an irregular two-dimensional array, but the low level implementation—which uses indirect arrays, unspecified preconditions, and array reshaping at call sites—-complicates analysis.

The second example, presented in Section 2.7.2, demonstrates how the same loop may encode two semantically different parallel operators: a map and a generalized reduction. Further difficulties refer to the compiler having to reverse engineer user's optimisations, such as mapping two semantically different arrays to the same memory block.

The third example, presented in Section 2.7.3, shows a loop that semantically corresponds to the sequential composition of a filter and a map. In this case the analysis has to be extended to (i) accommodate "conditional-induction variables" (CIV), and to (ii) reverse-engineer a composition of parallel-operators that is semantically equivalent with the original sequential loop.

2.7.1 Example 1: Difficult to Parallelise Map

Figure 13 shows the simplified version of loop `solvh_do20` from DYFESM benchmark of the PERFECT club benchmark suite. The aim is to prove that the (outermost) loop `solvh_do20` is parallel, in the sense that no dependencies exists between loop iterations—in essence the loop is semantically an irregular `map`, each iteration of it producing non-overlapping subsets of the elements of the `XE` and `HE` arrays.

The reasoning necessary for proving loop `solvh_do20` parallel is nontrivial [OR12], and in fact impossible using purely static techniques. The presence of exploitable parallelism is dependent on the statically unknown accesses using the indirect arrays `IA` and `IB`, which semantically model irregular 2D arrays.

The reasoning can proceed by looking at `XE` and `HE` as unidimensional arrays, and by observing that `XE` is written in subroutine `geteu` on all indexes belonging to interval `[1,16*NP]` and is read in `matmult` on indexes in `[1,NS]`.

```
SUBROUTINE solvh(HE,XE,IA,IB)
 DIMENSION  HE(32, *), XE(*)
 READ(*,*) SYM, NS, NP, N
 CCC    SOLVH_do20
 DO i = 1, N, 1
  DO k = 1, IA(i), 1
   id = IB(i) + k - 1
   CALL geteu    (XE, SYM,  NP)
   CALL matmult(HE(1,id),XE,NS)
   CALL solvhe   (HE(1,id), NP)
  ENDDO
ENDDO END

SUBROUTINE matmult(HE,XE,NS)
  DIMENSION HE(*), XE(*)

  DO j = 1, NS, 1
    HE(j) = XE(j)
    XE(j) = ...
ENDDO END
```

```
SUBROUTINE geteu(XE,SYM,NP)
 DIMENSION  XE(16,*)


  DO i = 1, NP, 1
   DO j = 1, 16, 1
    XE(j, i) = ...
   ENDDO
  ENDDO

END

SUBROUTINE solvhe(HE,NP)
 DIMENSION  HE(8, *)

 DO j = 1, 3, 1
  DO i = 1, NP, 1
   HE(j, i)=HE(j, i)+..
  ENDDO
ENDDO END
```

Figure 13: Simplified Loop SOLVH_DO20 from DYFESM.

Similarly, HE is written in matmult on all indexes in interval $[\tau+1,\tau+NS]$, and read and written in solvhe on a subset of indexes in interval $[\tau+1,\tau+8*NP-5]$, where τ=32*(id-1) is the array offset of parameter HE(1,id).

Read-After-Write independence of the outermost loop can be established by showing that the per-iteration read set of XE and HE are covered by their per-iteration write set. This corresponds to solving interval-inclusion equations

$$[1,NS] \subseteq [1,16*NP]$$

and

$$[\tau+1,\tau+8*NP-5] \subseteq [\tau+1,\tau+NS],$$

yielding predicate

$$NS \leq 16*NP \ \wedge \ 8*NP < NS+6$$

as sufficient condition for the flow independence of arrays XE and HE, respectively.

For *Write-After-Write independence*, one can observe that the per-iteration write set of array XE is invariant to the outermost loop, hence XE can be privatised and updated at the very end with the values written by the last iteration (i.e., static-last value).

The rationale for the write-after-write independence of array HE is more complicated: The write set of an iteration i of loop solvh_do20, denoted $\mathtt{WF}_i$, can be overestimated to the interval of indices:

```
[32*(IB(i)-1),32*(IB(i)+IA(i)-2)+NS-1]
```

The aim is to show that

$$\forall i \neq j,\ \mathtt{WF}_i \cap \mathtt{WF}_j = \emptyset$$

A sufficient condition [OR11] that works in practice is derived by checking the monotonicity of such intervals, i.e., the upper bound of $\mathtt{WF}_i$ is less than than lower bound of $\mathtt{WF}_{i+1}$, for all i. This results in the predicate

$$\wedge_{i=1}^{N-1} \mathtt{NS} \leq 32*(\mathtt{IB(i+1)}-\mathtt{IA(i)}-\mathtt{IB(i)}+1)$$

that verifies output independence under $O(\mathtt{N})$ runtime complexity.

2.7.2 Example 2: Reduction or Map?

Imperative techniques typically identify generalized-reduction patterns on an array (or scalar) variable A by checking that all its uses inside the target loop are of the form A[x] = A[x] ⊕.., where x is an arbitrary expression and ⊕ is a (known) binary associative operator.

The code below is an example of a generalized reduction, and can be parallelized by computing the changes to A locally on each processor and by merging (adding) them in parallel at the end of the loop.

```
DO i = 1, N, 1
    A(B(i)) = A(B(i)) + C(i)
ENDDO
```

However, if B is an injective mapping of indices, this treatment is unnecessary because each iteration reads and writes distinct elements of A, and thus each processor can work safely directly on the shared array A. In Futhark terms, in the latter case the loop is actually a composition of the parallel operators **map** and **scatter**.

Identifying the case when a generalized reduction is a map can require summarization of the iteration-wise read-write set of the target array A, denoted $\mathtt{RW}_i$ and checking that they do not overlap. This can be stated as the following equality:

$$\cup_{i=1}^{N} (\mathtt{RW}_i \cap \cup_{k=1}^{i-1} (\mathtt{RW}_k)) = \emptyset$$

Finally, sufficient conditions for identifying the **map** case can be extracted from this equation, for example by sorting the $\mathtt{RW}_i$ sets (as intervals), and then checking non-overlap. Such analysis is known as *runtime reduction* (RRED) in the literature [OR12], but it would not be necessary if the map-scatter parallelism would be explicitly expressed in the first place.

Another significant challenge to automatic parallelization are memory optimizations performed by the programmer. The classical example is privatization – an analysis that determines whether it is safe that the declaration of a variable is moved from outside to inside the loop. Another example is when two semantically different arrays are combined into the same array, as illustrated in the example below.

```
      DO i = 1, N, 1
S1:       A(i) = C(i)*2
S2:       A(B(i)) = A(B(i)) + C(i)
```

The statements `S1` and `S2` semantically build two different arrays: one which is constructed by a map operation, and another by a generalized-reduction operation.

To disambiguate such cases, compiler analysis needs to support an extended reduction pattern (EXT-RRED), where the target array `A` is allowed to be (only) written outside reduction statements, as long as the write accesses do not precede on any control-flow-graph path any reduction statement. Note that instances of EXT-RRED have non-empty per-iteration write-first ($\texttt{WF}_i$) and read-write ($\texttt{RW}_i$) sets and empty read-only set.

Enabling parallel execution in this case requires proving:

- Read-After-Write independence: $\cup_{i=1}^{N} (\texttt{WF}_i) \ \cap \ \cup_{i=1}^{N} (\texttt{RW}_i) = \emptyset$, and either
- Write-After-Write independence: $\cup_{i=1}^{N} (\texttt{WF}_i \cap \cup_{k=1}^{i-1} (\texttt{WF}_k)) = \emptyset$ or
- privatisation by static-last value: $\cup_{i=1}^{N} (\texttt{WF}_i) \subseteq \texttt{WF}_{i \leftarrow N}$

Loops MXMULT_DO10 and FORMR_DO20 that cover almost 55% of DYFESM's sequential runtime, exhibit both patterns discussed in this section.

2.7.3 Example 3: Conditional Induction Variables

Another challenge for automatic parallelisation are the so called "conditional-induction variables" (CIV) that represent scalars that do not form a uniform recurrence. For example, their increment may not be constant across the iteration space of the analysed loop.

```
civ@1 = 0
DO i = 1, N, 1
  civ@2=φ(civ@1, civ@4)
  IF C(i) .GT. 0 THEN
    DO j = 1, C(i), 1
      X(j+civ@2) = ...
    ENDDO
    civ@3 = C(i) + civ@2
  ENDIF
  civ@4 = φ(civ@3, civ@2)
ENDDO
civ@5=φ(civ@4, civ@1)
```

The code above shows a simplified version of loop `CORREC_do401` from BDNA benchmark, as an example of non-trivial loop that uses both CIV and affine-based subscripts. Variable `civ` is written directly in (gated) single-static-assignment (SSA) notation.

For example, statement `civ@2=`ϕ`(civ@1,civ@4)` has the semantics that variable `civ@2` takes either the value of `civ@1` for the first iteration of the loop or the value of `civ@4` for all other iterations.

The gist of the parallelisation technique [OR15] is to aggregate symbolically the CIV references on every control-flow path of the analyzed loop, in terms of the CIV values at the entry and end of each iteration or loop. The analysis succeeds if (i) the symbolic-summary results are identical on all paths and (ii) they can be aggregated across iterations in the interval domain. Parallelisation requires two main steps:

1 proving the absence of cross-iteration dependencies on array `X`.

2 computing in parallel the values of `civ` at the beginning of each iteration, i.e., the values of `civ@2`.

Step 1. The write accesses of the inner loop can be summarized by the interval

$$\mathtt{WF}_{inner-loop}\mathtt{=[civ@2,civ@2+C(i)-1].}$$

On the path on which condition `C(i).GT.0` holds, we have

$$\mathtt{civ@4=civ@3=civ@2+C(i)}$$

and the path summary is rewritten as

$$\mathtt{W}_i^{THEN} \mathtt{= [civ@2,civ@4-1].}$$

The other path neither updates `X` nor increments `civ`, hence it can use the same symbolic summary

$$\mathtt{W}_i^{ELSE} \mathtt{= [civ@2,civ@4-1]} = \emptyset$$

because on that path `civ@2=civ@4>civ@4-1`, and an interval having its lower bound greater than its upper bound is considered empty.

It follows that the summaries of the `THEN` and `ELSE` branches can be unified; the iteration summary being

$$W_i \ = \ \texttt{[civ@2,civ@4-1]}.$$

Loop independence can now be proven by verifying the set equation

$$\cup_{i=1}^{N} (\texttt{W}_i \ \cap \ \cup_{k=1}^{i-1} (\texttt{WF}_k)) = \emptyset,$$

which holds because $\cup_{k=1}^{i-1}$ `(WF`$_k$`)` can be symbolically computed to be `[1, civ@2`^{i}`-1]` by using:

- the implicit invariant `civ@4`$^{i-1}$ `= civ@2`^{i}, i.e., the `civ` value at an iteration end is equal to the `civ` value at the entry of the next iteration, and
- the monotonicity of the `civ@2` values (because they are incremented by `C(i)` only when `C(i)` is positive).

Step 2. While the accesses to `X` have been disambiguated, `civ` still remains the source of cross iteration dependences.

The CIV values at the beginning of each iteration can be computed in parallel by the following technique: First, the slice that computes `civ` in an iteration is extracted and it is checked that `civ` appears only in reductions statement in it. Second, the `civ@2` is initialized (in the slice) to the neutral-element of the reduction operator. Third, the obtained slice is mapped across the original iterations space, and the result is subjected to an exclusive scan.

The Futhark code below illustrates the result of this technique on our example[9]:

```
scan (+) 0 (map (\i -> if C[i] > 0 then C[i] else 0)
                [1...N])
```

We conclude by remarking that in addition to this "heroic" analysis, which is necessary for proving the absence of cross-iteration dependences for array `X` that uses CIV-based accesses, the compiler also had to re-engineer the computation of `civ` values from inherently sequential to parallel. We argue that, if parallel execution is desired, the code should have been written from the very beginning in terms of data-parallel operators rather than sequential loops.

[9]Except that **`scan`** in Futhark is inclusive, not exclusive.

2.8 Related Work on Data-Parallel Languages

Fortunately, not all work on parallelism involves Fortran 77. There is a rich body of literature on embedded array languages and libraries targetting GPUs. Imperative solutions include Copperhead [CGK11], Accelerator [TPO06], and deep-learning DSLs, such as Theano [Ber+10] and Torch [CKF11].

Purely functional languages include Accelerate [McD+13], Obsidian [CSS12], and NOVA [Col+14]. While these languages differ significantly, one common limitation is a lack of support for arbitrary nested regular parallelism, as well as explicit indexing and efficient sequential code inside their parallel constructs.

A number of dataflow languages aim at efficient GPU compilation. StreamIt supports a number of static optimizations on various hardware, for example, GPU optimizations [Hor+11] include memory-layout selection (shared/global memory), resolving shared-memory bank conflicts, increasing the granularity of parallelism by vertical fusion, and untilizing unused registers by software prefetching and loop unrolling, while multicore optimizations [GTA06] are aimed at finding the right mix of task, data and pipeline parallelism.

Recent work has been done on ameliorating the downsides of full flattening. These include data-only flattening [Ber+13], and *streaming* [MF16]. The streaming approach uses the type system to classify sequences into arrays and *streams*, where the latter do not support random access, but only structured operations such as `map`, `reduce`, and `scan`. This allows an incremental dataflow-oriented execution model that avoids the explosion in memory usage that often accompanies full flattening. Despite the similarity in naming, Futhark's **`stream_map`** and **`stream_red`** operators do not directly support this notion of streaming, as they do not ban random access into arbitrary arrays (including the array being "streamed") from within the chunk function.

Chapter 3

An Array Calculus

The theoretical inspiration behind Futhark are the list homomorphisms described by Bird and Meertens [Bir89], realised in the form of purely functional parallel second-order array combinators (SOACs). Their rich equational theory for semantics-preserving transformations are employed in Futhark for fusion, streaming, and flattening of parallelism. This chapter discusses a simple array combinator calculus that is used to convey the intuition behind certain useful transformations on arrays and, more importantly, functions that operate on arrays. In Section 4.5 we shall also see that the primitives defined here have an efficient implementation on the GPU.

The calculus presented here is simpler than the full Futhark programming language, yet also more flexible. While the names of the combinators intentionally resemble those of Futhark (both the source language and the core language we will see later in the thesis), they are not fully identical.

3.1 Array Combinator Calculus

The array combinator calculus, a modified lambda calculus, describes terms that operate on array values. The syntax is summarised in Figure 14. We make a distinction between *ground values* v, which cannot contain functionals, and *terms* e, which can. An array value is written as a comma-separated sequence of values enclosed in square brackets, as in $[v_1, v_2, v_3]$. Arrays may contain scalar values written as $\mathtt{t}$ (which are

$$
\begin{array}{llr}
\alpha, \beta, \tau & ::= & \mathtt{t} \mid (\tau_1, \ldots, \tau_n) \mid [n]\tau \\
v & ::= & \mathtt{v} \mid (v_1, \ldots, v_n) \mid [v_1, \ldots, v_n] \\
\hat{\alpha}, \hat{\beta}, \hat{\tau} & ::= & \tau \mid \hat{\alpha} \to \hat{\beta} \mid \Pi n.\hat{\tau} \\
e & ::= & \mathtt{v} \mid (e_1, \ldots, e_n) \mid [e_1, \ldots, e_n] \mid e_1\ e_2 \mid \lambda(x_1, \ldots, x_n).e
\end{array}
$$

Figure 14: Syntax of the Array Combinator Calculus.

not important for the calculus), other arrays, or tuples of values. A tuple is written as a comma-separated sequence of values enclosed in parentheses.

Array values are classified by array types. We write $[n]\tau$ to denote the type of arrays with n elements of type τ. This effectively bans irregular arrays, as there is no way to provide n, m such that the value $[[1,2],[3]]$ is classified by a type $[n][m]\texttt{i32}$.

The type of a tuple is written as the types of its components separated by commas and enclosed in parentheses. Primitive (non-array) types are written as `t`. Similarly to the MOA [HM93] array formalism, we treat the curry/uncurry-isomorphic types $[m]([n]\tau) \cong [m \times n]\tau$ as interchangeable. This isomorphic treatment is justified because both streaming and indexed access to either type can be efficiently implemented without explicitly applying the isomorphism and materializing (storing) the result first. This is different from the Futhark source language, where for example $[1][n]\tau$ and $[n][1]\tau$ are distinct types. Likewise, a singleton array $[1]\tau$ is distinct from a scalar of type τ.

Higher-order types $\hat{\tau}$ for classifying functions are a superset of value types. Specifically, we do not permit first class functions in the calculus, although we do permit functional arguments in function types. Function types support both ordinary value abstraction $\hat{\alpha} \to \hat{\beta}$, as well as *size abstraction*, written $\Pi n.\hat{\tau}$. This allows us to specify a function that can operate on an array of any size, and returns an array of that same size, which could have the following type:

$$\Pi n.[n]\tau \to [n]\tau$$

The size of an array returned by a function must be expressible in terms of the size of its input. This restriction could be lifted by supporting $\exists$-quantification in types, but we omit this for the array calculus, although it is supported in the Futhark core language (Chapter 6).

We treat the `zip`/`unzip`-isomorphic types $[n](\tau_1, \ldots, \tau_k) \cong ([n]\tau_1, \ldots, [n]\tau_k)$ as interchangeable in any context. This is justified as any array of tuples can be converted into a corresponding tuple of arrays, as shown in Figure 15. The function $C_{\mathcal{V}}(v)$ rewrites a value v such that no arrays of tuples appear. Likewise, the function $C_{\mathcal{T}}(\tau)$ rewrites a type. Intuitively, the rewriting considers an array of tuples as an augmented array with an extra dimension, followed by transposing out that extra dimension. Actually constructing this array would not be well-typed, as all elements of an array must have the same type, whereas tuples can contain elements of differing types.

3.2 Basic SOACs

We first describe the basic SOACs and later introduce *streaming combinators*. The basic SOACs include (i) map, which constructs an array by applying its function argument to each element of the input array, (ii) reduce, which applies a binary-associative operator $\oplus$ to all elements of the input, and (iii) scan, which computes the

$$C_{\mathcal{V}}([(v_{(1,1)}, \ldots, v_{(1,m)}), \ldots, (v_{(n,1)}, \ldots, v_{(n,m)})]) =$$
$$(C_{\mathcal{V}}([v_{(1,1)}, \ldots, v_{(n,1)}]), \ldots, C_{\mathcal{V}}([v_{(1,m)}, \ldots, v_{(n,m)}]))$$

$$C_{\mathcal{T}}([k](\tau_1, \ldots, \tau_n)) =$$
$$(C_{\mathcal{T}}([k]\tau_1), \ldots, C_{\mathcal{T}}([k]\tau_n))$$

Figure 15: Converting arrays of tuples to tuples of arrays. The function $C_{\mathcal{V}}$ transforms values, and $C_{\mathcal{T}}$ transforms types. The equalities are applied recursively until a fixed point is reached. We claim the property that if $v : \tau$, then $C_{\mathcal{V}}(v) : C_{\mathcal{T}}(\tau)$.

sums of all prefixes of the input array elements. Their types and semantics are shown below:

$$\text{map} \ : \ (\alpha \to \beta) \ \to \ \Pi n.[n]\alpha \ \to [n]\beta$$
$$\text{map} \ f \ [a_1, \ldots, a_n] \ = \ [f \ a_1, \ldots, f \ a_n]$$
$$\text{reduce} : (\alpha \to \alpha \to \alpha) \ \to \ \alpha \ \to \ \Pi n.[n]\alpha \to \alpha$$
$$\text{reduce} \ \oplus \ 0_{\oplus} \ [a_1, \ldots, a_n] \ = \ 0_{\oplus} \oplus a_1 \oplus \ldots \oplus a_n$$
$$\text{scan} \ : \ (\alpha \to \alpha \to \alpha) \ \to \ \alpha \ \to \ \Pi n.[n]\alpha \ \to \ [n]\alpha$$
$$\text{scan} \ \oplus \ 0_{\oplus} \ [a_1, \ldots, a_n] \ = \ [\text{reduce} \ \oplus \ 0_{\oplus} \ [a_1], \ldots, \text{reduce} \ \oplus \ 0_{\oplus} \ [a_1, \ldots, a_n]]$$

We can view the Πn notation as indicating where the size n becomes fixed; it indicates, for instance, that we can partially apply map to a function and apply the resulting function to arrays of different sizes.

The SOAC semantics enables powerful rewrite rules. For example, mapping an array by a function f followed by mapping the result with a function g gives the same result as mapping the original array with the composition of f and g:

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

Applied from left-to-right and from right-to-left this rule corresponds to producer-consumer (*vertical*) fusion and fission, respectively. *Horizontal* fusion/fission refers to the case when the two maps are independent (i.e., not in any producer-consumer relation), as in the equation below:

$$(\text{map} \ f \ x, \text{map} \ g \ y) \ \equiv \text{map} \ (\lambda(a, b).(f \ a, g \ b)) \ (x, y)$$

The rest of this chapter shows how map and reduce are special cases of a more general bulk-parallel operator named redomap, which (i) can represent (fused) compositions of map and reduce operators and, as such, (ii) can itself be decomposed into a map-reduce composition. Similarly, we introduce the parallel operator sFold, which generalizes Futhark's streaming operators.

3.2.1 Notation

We denote array concatenation by # and the empty array by ϵ; $\text{inj}(a)$ is the single-element array containing a. A *partitioning* of an array v is a sequence of arrays $v_1, \ldots, v_k$ such that $v_1 \# \ldots \# v_k = v$. Given operations f and g, their *product* $f * g$ is defined by component-wise application, i.e., $(f * g)(x) = (f\ x, g\ x)$.

3.2.2 The Parallel Operator redomap

Many array operations are *monoid homomorphisms*, which conceptually allows for splitting an array into two parts, applying the operation recursively, and combining the results using an associative operation $\oplus$. For example, a map−reduce composition can be formally transformed, via the list homomorphism promotion lemma [Bir87], to an equivalent form:

$$\text{reduce}\ \oplus\ 0_\oplus \circ \text{map}\ f \equiv \text{reduce}\ \oplus\ 0_\oplus \circ \text{map}\ (\text{reduce}\ \oplus\ 0_\oplus \circ \text{map}\ f) \circ \text{split}_p$$

where the original array is partitioned into p chunks. Operationally, we could imagine that p processors will each perform the map − reduce composition sequentially, followed by a parallel combination of the per-processor results. Hence, the *inner* map − reduce can be written as a left-fold

$$\text{reduce}\ \oplus\ 0_\oplus \circ \text{map}\ f \equiv \text{reduce}\ \oplus\ 0_\oplus \circ \text{map}\ (\text{foldl}\ g\ 0_\oplus) \circ \text{split}_p$$

for an appropriately defined function g, which must be semantically equivalent to

$$g = \lambda x y \to x \oplus (f\ y)$$

but may have a more efficient implementation, typically by combining $\oplus$ and f.

Every monoid homomorphism can thus be uniquely determined by $(\oplus, 0_\oplus)$ and a function g. The combinator redomap[1] expresses all such homomorphisms:

$$\begin{aligned}&\text{redomap} : (\alpha \to \alpha \to \alpha, \alpha) \to (\beta \to \alpha) \to \Pi n.([n]\beta \to \alpha)\\ &\text{redomap}\ (\oplus, 0_\oplus)\ g\ [b_1, \ldots, b_n] = 0_\oplus \oplus (g\ 0_\oplus\ b_1) \oplus \ldots \oplus (g\ 0_\oplus\ b_n)\end{aligned}$$

The redomap combinator can decompose previously seen SOACs:

$$\begin{aligned}\text{map}\ g &= \text{redomap}\ (\#, \epsilon)\ (\text{inj} \circ g)\\ \text{reduce}\ (\oplus, 0_\oplus) &= \text{redomap}\ (\oplus, 0_\oplus)\ \text{id}\end{aligned}$$

and redomap can itself be decomposed by the equation

$$\text{redomap}\ (\oplus, 0_\oplus)\ g\ =\ \text{reduce}\ (\oplus, 0_\oplus) \circ \text{map}\ (g\ 0_\oplus).$$

[1] The name is a portmanteau of reduce ∘ map.

3.2.3 Streaming combinators.

A key aspect of Futhark is to *partition* implementations of redomap, such that they partition vectors into *chunks* before applying the operation on the chunks individually and eventually combining them:

$$\begin{aligned}&\text{sFold} : (\alpha \to \alpha \to \alpha) \to (\Pi m.([m]\beta \to \alpha)) \to \Pi n.[n]\beta \to \alpha\\&\text{sFold}\ (\oplus)\ f\ (v_1 \# \ldots \# v_k) = (f\ \epsilon) \oplus (f\ v_1) \oplus \ldots \oplus (f\ v_k)\end{aligned}$$

Because a vector can have multiple partitions, sFold is well-defined—it gives the same result for all partitions—if and only if f is equivalent to a redomap with $\oplus$ as combining operator. Futhark assumes such properties to hold; they are not checked at run-time, but the responsibility of the programmer.

The streaming combinators permit Futhark to choose freely *any* suitable partition of the input vector. In the source language, Futhark uses specialized versions of sFold:

$$\begin{aligned}\texttt{stream\_map}\ f &= \text{sFold}\ (\#)\ f\\\texttt{stream\_red}\ (\oplus)\ f &= \text{sFold}\ ((\oplus) * (\#))\ f\end{aligned}$$

Fusion and fission transformations are based on the universal properties of redomap (and sFold); for example, horizontal (parallel) fusion is expressed by the "banana split theorem" [MFP91], read as a transformation from right to left:

$$\text{redomap}\ ((\oplus, 0_\oplus) * (\otimes, 0_\otimes))\ (f, g) = (\text{redomap}\ (\oplus, 0_\oplus)\ f, \text{redomap}\ (\otimes, 0_\otimes)\ g)$$

The map-map rule $\text{map}\ (g \circ f) = \text{map}\ g\ \circ\ \text{map}\ f$ is the functorial property of arrays; it is used for fusion from right to left and eventually, as a fission rule, from left to right as part of flattening nested parallelism (see Chapter 8). The flattening rule $\text{map}(\text{map} f) \cong \text{map}\ f$ eliminates nested parallel maps by mapping the argument function over the product index space (an isomorphism modulo the curry/uncurry isomorphism). Finally, sequential (de)composition of the discussed SOACs can be similarly reasoned in terms of the general iterative operator sFold; for example:

$$\begin{aligned}\text{map} f &= \text{sFold}\ (\#)\ \epsilon\ (\text{map}\ f)\\\text{reduce} \oplus 0_\oplus &= \text{sFold}\ \oplus\ 0_\oplus\ (\text{reduce} \oplus 0_\oplus)\end{aligned}$$

Chapter 4

Parallelism and Hardware Constraints

There exists a large diversity of parallel machines—far too many to list exhaustively, and far too different in capability and performance characteristics for one set of program transformations and optimisations to apply universally. In this thesis, we focus our discussion on one type of architecture: the modern *Graphics Processing Unit* (GPU), which can be used for non-graphics computation under the term *General-Purpose GPU computing* (GPGPU). Specifically, we focus on the *Single-Instruction Multiple-Thread* (SIMT) model used by NVIDIA, and also seen in recent chips from AMD and Intel.

This chapter presents a simplified abstract GPU machine model that we will use to justify the design of the Futhark compiler. In particular, a machine model is needed to argue for the optimising transformations that we perform. The model defined here serves this purpose. As an example, Section 4.3.1 describes how matrix transposition is implemented efficiently on GPUs. This is crucial, since Futhark uses transpositions to optimise for spatial locality (Section 9.1). Section 4.5 shows how selected important primitive constructs from the array calculus (Chapter 3) can be mapped to GPU code. This is of particular interest since Chapter 8 shows how nested parallelism is transformed into constructs similar to those in the array calculus.

When we use the term "GPU", we will refer to the model defined herein. Our chosen terminology is primarily taken from OpenCL, an open standard for programming *accelerator devices*, of which GPUs are one example. Unfortunately, NVIDIAs API for GPGPU computing, CUDA, uses the same terms in some cases, but with different definitions. Section 4.4 describes how our GPU model maps to CUDA terms, and how our simplifications map to real hardware.

```
thread(in, out):
  gtid      <- get_global_id()
  x         <- in[gtid]
  y         <- x + 2
  out[gtid] <- y
```

Figure 16: Each thread reads an integer, adds two, and writes the result back to some other memory location

4.1 Kernels and the Thread Space

A GPU is an accelerator device attached to a conventional CPU-based computer, which is called the *host* system. A GPU program is called a *kernel*, and is launched by the host. A running kernel comprises some quantity of independent threads, each of which is executing the same sequential program. When a kernel is launched, the host system dictates how many threads are to be used, and how they are to be organised. These parameters can vary from one launch of the same kernel to the next. Furthermore, each thread may also be passed some parameters containing values or memory addresses – the same for every thread.

The threads are organised into equally sized *workgroups* (or just *groups*), within which communication between threads is possible. The total number of threads launched is thus always a multiple of the group size. The hardware limits both the maximum group size (typically 1024 or less), as well as the maximum number of groups, although the latter is rarely relevant in practice. For kernels in which each thread is fully independent (what we call *map-parallelism*), the group size is mostly irrelevant.

As each thread runs a sequential program, the number of threads launched for a kernel is the sole way that we can express parallelism. Nested parallelism is not supported in this model.

Each thread in a running kernel can be uniquely identified by its *global thread ID* ($gtid$). Futhermore, each thread belongs to a group, identified by a *group id* (gid) that is shared by all threads in the group. Within a group, each thread has a *local thread ID* ($ltid$) that is unique only within the workgroup. If the group size is $gsize$, then

$$gtid = gid \cdot gsize + ltid$$

holds for every thread. The thread index space is single-dimensional, and each ID is a single integer. An example in imperative pseudocode is shown in Figure 16, which demonstrates how threads use their identifiers to read input and write output. The `in` and `out` parameters to the thread are addresses of arrays in memory.

```
thread(arr):
  gtid <- get_global_id()
  x    <- arr[gtid]
  if (x < 0):
    x <- -x
  arr[gtid] <- x
```

Figure 17: A kernel that computes the absolute value in-place on the elements of an array.

4.2 The GPU Model

When we launch a kernel on the GPU, we specify a group size and the desired number of groups. This does not mean that every thread is actually running concurrently. The GPU hardware has some quantity of resources available – for this discussion, most notably registers and local memory space. A group requires some kernel-dependent amount of registers for each of its threads, as well as some (potentially zero) amount of local memory. The GPU may be able to fit anywhere from zero to all groups concurrently. Every thread in a group must be able to fit on a GPU simultaneously - fractional groups are not permitted.

If not all groups fit at the same time, then as many as possible will be launched, with remaining groups being launched once earlier ones finish (the order in which groups are launched is undefined). This is another reason why communication between groups is difficult: unless we have detailed low-level information about both GPU and kernel, we cannot know whether the group we wish to communicate with has even been launched yet! In principle, the GPU may even be running groups from different kernel launches concurrently, perhaps even belonging to different users on the host system. Due to these difficulties, we generally take the view that communication is only possible within a workgroup.

GPUs execute neighbouring threads in lockstep. The threads within a group are divided into *warp*s, the size of which is hardware-dependent. These warps form the basis of execution and scheduling. All threads in a warp execute the same instruction in the same clock cycle. Branches are handled via *masking*. All threads in a warp follow every conditional branch for which the condition is true for at least a single thread in the warp. For those threads where the condition is false, a flag is set that makes the GPU ignore the instructions being executed. This means that we cannot hide expensive but infrequent computation behind branches, the way we typically to on CPUs: if just one thread has to enter the expensive region, then we pay the cost for every thread in the warp. An example of a kernel that contains branches is shown in Figure 17. Even if the condition on line 4 is false for a given thread, it may still execute line 5. But if so, the mask bit will be set, and the write to `x` will be ignored.

4.3 Memory Spaces

The GPU has several kinds of memory, each with its own properties and performance considerations. As the limiting factor in GPU performance is typically memory access times, proper use of the various kinds of memory is important. The following kinds of memory are used in our GPU model:

Registers, which are private to each thread. Registers are scarce but fast, and while they can be used to hold small arrays, they are typically used to hold scalar values. Overuse of registers can lead to fewer threads being able to run concurrently, or perhaps to the kernel being unable to run at all. The number of registers used by a thread is a function solely of its code, and cannot depend on kernel-launch parameters.

Local memory, which is fast but size-constrained on-chip memory, and is shared by all threads in a group. When a kernel is launched, some quantity of local memory can be requested per group (same amount for each). The contents of a local memory buffer becomes invalidated once its associated group terminates, and hence cannot be used for passing results back to the CPU, or to store data from one kernel launch to the next.

Global memory, which is large off-chip memory, typically several GiB in size. Although much slower than registers or shared memory, global memory is typically still much faster than CPU RAM. The contents of global memory are visible to all groups, and persists across kernel launches. Thus, global memory is the only way to pass initial data to a kernel, and for a kernel to return a result. Global memory can be read from or written to from the CPU, although at much lower speeds than the GPU itself is able to.

Global memory does not have the many layers of caching that we are used to from the CPU. Instead, we must make sure to only access global memory with efficient access patterns (see Section 4.3.1).

Global memory must be allocated by the host prior to kernel execution. Memory allocation is not possible from within a running kernel.

Threads communicate with each other by reading and writing values to memory (typically, local memory). Since GPUs use a weakly consistent memory model, we must use *barrier instructions* to ensure that the memory location we wish to read from has already been written to by the responsible thread. When a thread executes a barrier instruction, it will wait until every other thread in the group has also reached the barrier instruction, at which point they will continue execution. The fact that barriers have an effect only within a group is why we say that communication can only happen between threads in the same group.

4.3.1 Coalesced Memory Accesses

The connection from GPU to global memory is via a fast and wide bus that is able to fetch several words in a single transaction. As an example, many NVIDIA hardware GPus uses a 512-bit bus that can fetch 16 32-bit words in one transaction, but only if the words are stored adjacent in memory and do not cross a cache line. We will ignore the cache line concern in the following, as it is a secondary effect.

This memory bus behaviour has important performance implications. Fetching 16 neighbouring words can be done in one memory transaction, whilst fetching 16 widely spaced words will require a transaction per word, thus utilising only a sixteenth of available memory bandwidth. As the performance of many GPU programs depends on how quickly they can access memory (they are *bandwidth-bound*), making efficient use of the memory bus has high priority.

When a warp issues a memory operation such that adjacent threads access adjacent locations in memory, we say that we have a *coalesced memory access*. This term comes from the notion that several memory operations *coalesce* into one. On some GPUs, the addresses accessed by the threads in a warp must be ascending by one word per thread, but on newer GPUs, it can be any permutation we wish, as long as every address falls within the same 512-bit memory block. On an NVIDIA GPU, in the best case, a 32-thread warp can store 32 distinct 32-bit words using two memory transactions.

The optimal access pattern is thus a bit counter-intuitive compared to what we are used to from CPU programming. On the GPU, a single thread should access global memory with a stride–something that is known to exhibit bad cache behaviour on a CPU. An example is shown on Figure 18. In both of the kernels shown, each thread sums `n` elements of some array. The difference lies in the access pattern. On Figure 18a, when thread j accesses the element at index $j \cdot n + i$, thread $j + 1$ will be accessing the element at index $(j + 1) \cdot n + i$. Unless n is small, these two addresses are far from each other, and will require a memory transaction each. In contrast, Figure 18b shows a program where thread j accesses index $i \cdot n + j$ while thread $j + 1$ accesses index $i \cdot n + j + 1$. These are adjacent locations in memory and can likely be fetched in just one memory operation. In practice, the latter program will run significantly faster – a factor 10 difference is not at all unlikely.

GPUs have little cache in the way we know from CPUs. While we can use local and private memory as a manually managed cache, most of our memory performance will come from rapidly reading and writing from global memory via coalesced memory operations.

A coalescing example: transposition

As an example of how coalescing can be resolved in a nontrivial case, this section will present how two-dimensional matrix transposition can be implemented in our

```
thread(in, out, n):
  gtid <- get_global_id()
  sum <- 0
  for i < n:
    sum <- sum + in[gtid*n + i]
  out[gtid] <- sum
```

(a) Uncoalesced access.

```
thread(in, out, n):
  gtid <- get_global_id()
  sum <- 0
  for i < n:
    sum <- sum + in[i*n + gtid]
  out[gtid] <- sum
```

(b) Coalesced access.

Figure 18: Examples of programs with uncoalesced and coalesced memory accesses.

```
thread(in, out, n, m):
  gtid <- get_global_id()
  i <- gtid / n
  j <- gtid % m

  index_in <- i * n + j
  index_out <- j * m + i

  out[index_out] <- in[index_in]
```

Figure 19: Naive GPU implementation of matrix transposition.

GPU model. The basic idea is to launch a kernel that contains one thread for every element in the input matrix. For simplicity, we assume that the total number of input elements is divisible by our desired workgroup size.

A naive attempt can be seen in Figure 19. Because the GPU model does not have a notion of multi-dimensional arrays, we assume the nominally $n \times m$ matrix is represented as a flat array in row-major representation.

The write to the `out` array is fine, as it is coalesced between neighbouring threads.

gtid	i	j	index_in	index_out
0	0	0	0	0
1	0	1	1	100
2	0	2	2	200
3	0	3	3	300
			⋮	
100	1	0	100	1
101	1	1	101	101
102	1	2	102	201

(a) Tabulation of computed indexes for the transposition kernel on Figure 19. We assume $n = m = 100$.

gtid	g_i	g_j	l_i	l_j	index_in	index_out
0	0	0	0	0	0	0
1	0	1	0	1	1	1
2	0	2	1	0	2	2
3	0	3	1	1	3	3
				⋮		
100	1	0	0	0	100	100
101	1	1	0	1	101	101
102	1	2	1	0	102	102

(b) Tabulation of computed indexes for the transposition kernel on Figure 20. We assume $n = m = 100$, and `TILE_SIZE` = 2 (this is an implausibly small number, 16 is more realistic).

Table 2: Tables of indices computed for the two transposition kernels.

However, the read from `in` is problematic. Suppose $n = m = 100$. If we perform the index arithmetic by hand, we obtain the threads and indexes shown on Table 2a.

The solution is to perform a *tiled transposition*. Every workgroup will read a chunk of the input array into a local memory array, and then write back that chunk in transposed form. The trick is that the element that a given thread reads may not be the same as the element that it writes. We could say that we perform the transposition in local memory, rather than global memory. The approach is shown on Figure 20. We assume some constant `TILE_SIZE`, and that the workgroup size is exactly `TILE_SIZE` · `TILE_SIZE`. We further assume the presence of a local memory array `block` with room for one element for every thread in the workgroup. Recall that local memory arrays are local to the workgroup, and visible by all threads in the workgroup.

We use a memory barrier between reading the input and writing the output. This is necessary to ensure that the element we read from `block` has been written by the responsible thread, as GPUs do not guarentee execution order between warps.

```
local block[TILE_SIZE*TILE_SIZE]

thread(in, out, n, m):
  gtid <- get_global_id()
  g_i <- gtid / n
  g_j <- gtid % m

  ltid <- get_local_id()
  l_i <- ltid / TILE_SIZE
  l_j <- ltid % TILE_SIZE

  index_in <- g_i * m + g_j
  index_out <- g_i * n + g_j

  block[l_j*TILE_SIZE + l_i] <- in[index_in]

  barrier()

  out[index_out] <- block[l_i*TILE_SIZE + l_j]
```

Figure 20: A GPU implementation of matrix transposition that ensures coalesced access to memory. We assume that the workgroup size is `TILE_SIZE` $\cdot$ `TILE_SIZE`.

The coalesced kernel is very efficient on current GPU hardware, and is in fact not much slower than an ordinary memory copy. Thus we can treat transposition as a fast primitive. It is also not hard to extend the transposition kernel such that it transposes not just a single two-dimensional array, but rather transposes the inner two dimensions of a three-dimensional array. This is useful for implementing Futhark's **`rearrange`** construct. We have two final remarks on transposition:

1. For this kernel, we assume that n and m are divisible by `TILE_SIZE` in both dimensions. This assumption can be lifted by the liberal application of bounds checks before accessing global memory. However, such checks may make the kernel inefficient for pathological cases where n or m is very small (corresponding to very "fat" or "skinny" matrices), as many threads per workgroup will be idle. Specialised kernels should be used for these cases.

2. The way the kernel on Figure 20 accesses local memory is susceptible to *bank conflicts*, which is a condition where several threads in the same memory clock cycle attempt to access different addresses within the same memory bank. We do not address bank conflicts further here, except to note that they are solvable by padding the `local` array so that it has size `TILE_SIZE*(TILE_SIZE+1)`.

Our term	CUDA term
Workgroup	Block
Private memory	Local memory
Local memory	Shared memory

Table 3: Thesis terms to CUDA terms.

4.4 The Model in the Real World

The model presented in this chapter is intended as a tool for justifying the transformations performed by the Futhark compiler. But as our eventual goal is efficient execution on real silicon, the model is useless if it does not permit an efficient mapping to reality. In this section, I will explain how our GPU model maps to the popular CUDA [17] model supplied by NVIDIA. One issue is that of terminology. This thesis uses terms from the OpenCL standard, which overlaps in incompatible way with terms defined by CUDA. Table 3 summarises the differences. In this section we will continue using the same terms as in the rest of the thesis, even when talking about CUDA.

The most significant difference between our GPU model and CUDA is the dimensionality. In CUDA, groups and threads are organised into a three-dimensional *grid* (x, y, z), while our model contains only a single dimension. This is primarily a programming convenience for problems that can be decomposed spatially. However, the grid has hardware-specific size constraints: on recent NVIDIA hardware, the maximum number of threads in a workgroup in the x and y dimensions can be at most 1024, and in the z dimension at most 64. Also, the total number of threads in a workgroup may not exceed 1024.

The one-dimensional space of our GPU model can be mapped to CUDA by only using the x dimension, and leaving the y and z dimensions of size 1. However, CUDA imposes more restrictions: we can have at most $2^{16} = 65535$ workgroups in one dimension. Usually this will not be a problem, as no GPU needs anywhere this many threads to saturate its computational resources. If it were to become a problem in the future, we could simply map the flat group ID space onto CUDAs three-dimensional space by computing, for any gid:

$$x = \frac{gid}{2^{16}}, \quad y = gid \bmod 2^{16}, \quad z = 0$$

In general, flat and nested spaces can be mapped to each other at the cost of a modest number of division and remainder operations. The only place where the layout of the thread space has semantic or performance consequences is within workgroups. As the size limit for these (1024) is the same as the maximum number of threads along the x dimension, we can always treat them as one-dimensional.

In the extreme case, we can also virtualise workgroups by adding a loop to each physical workgroup. This loop runs the body multiple times with different values for

gid and $gtid$. This works because we assume communication between workgroups to be impossible, and hence the order of their execution to be irrelevant.

4.4.1 Excluded Facilities

Real GPU hardware possesses some capabilities that are not represented in our model. Some of these are unimportant, but others do offer performance opportunities that are for this work lost to us.

Atomic Operations

One important missing facility is atomic global memory operations, by which a thread can modify a location in memory in a race-free manner. The hardware supports a fixed number of atomic operators, such as addition, subtraction, exchange, or taking minimum and maximum. Crucially, there is also an atomic compare-and-swap instruction, on top of which other synchronisation primitives could be built [Fra04].

Using atomic addition, we could represent a summation kernel that needs only one kernel launch instead of two (see Section 4.5 for a general treatment of how reduction is implemented). The reason we do not exploit these atomic operations is due to their lack of generality. To the Futhark compiler, there is nothing special about a summation: it is treated like any other commutative reduction. Could the code generator, or some intermediate optimisation pass, employ pattern matching to recognise known patterns that have particularly efficient realisations? Certainly, and perhaps we will do so in the future, but for the present work we have chosen to focus on the general case, in the interest of simplicity. Using compare-and-swap, it is feasible to implement single-state reduction for arbitrary operators, but it is unclear whether it is worth the cost in complexity, as the memory traffic would not be smaller than for the two-stage kernel. The primary savings would be in avoiding the overhead of launching the second (single-group) kernel, which by itself is not all that expensive.

Atomic operations are most useful when used to implement communication between workgroups. An implementation of prefix sum (`scan`) in CUDA has been developed which performs just one full pass over the input [MYB16]. In contrast, the best version available to us requires two. However, the single-stage `scan` implementation depends on intimate knowledge about the target hardware—in particular, launching no more groups than will fit concurrently. This is not just a question of knowing the hardware, but also its users. If some other program is also using the GPU, we will have room for fewer concurrent groups than expected, potentially resulting in deadlock. We have judged the fragility of such code unacceptable for the output of a compiler, especially since the compiler is intended to shield its users from having to possess deep hardware knowledge. In contrast to languages that employ full flattening (such as NESL [Ble96]), Futhark performance is also less dependent on `scan` performance.

More Memory Spaces

Real GPUs support more kinds of memory than we have included in our GPU model:

Constant memory is a size-limited read-only on-chip cache. When we declare an array input to a kernel function, we can mark it as *constant*, which will cause it to be copied to the cache just before the kernel begins execution. Constant memory is useful when many threads are reading from the same memory in the same clock cycle, as accessing the constant cache is significantly faster (in terms of latency) than accessing global memory.

Unfortunately, this facility is difficult to exploit for a compiler. The reason is that the size of the constant cache is quite small—usually not more than 64KiB—and the kernel launch will fail if the arrays passed for constant parameters do not fit in the cache. In general, the specific sizes of arrays is not known until runtime. Thus, the choice of whether to use constant memory is a compile-time decision, but the validity of the decision can only be known at run-time.

Texture memory is physically global memory, but marked as read-only, and *spatially cached*, and, in contrast to other memories, multi-dimensional and with assumptions on the type of elements stored. Texture memory is useful for access patterns that exhibit spatial locality in a multi-dimensional space. For example, for a two-dimensional array `a`, the elements `a[i,j]` and `a[i+1,j+1]` are distant from each other no matter whether `a` is represented in row- or column-major form, but they are spatially close in a two-dimensional space.

Texture memory is cached in such a way that elements are stored together with spatially nearby elements. When texture memory is allocated, we must specify its dimensionality, which in extant hardware must be of 1, 2, or 3 dimensions. There are also restrictions on the size of individual dimensions. On an NVIDIA GTX780, 1D buffers are limited to 2^{27} elements, 2D buffers to $2^{16} \cdot 2^{16}$ elements, and 3D buffers to $2^{12} \cdot 2^{12} \cdot 2^{12}$.

Texture memory also supports other convenient operations with full hardware-acceleration. For example, out-of-bounds accesses can be handled transparently via index-wraparound, or by producing a specified "border" value. Automatic linear interpolation between neighbouring values is also provided. However, for non-graphics workloads, the spatial locality caching model is the most interesting feature. Unfortunately, it is difficult for a compiler to exploit. The size limitations are particularly inconvenient for "skinny" arrays that are much larger in one dimension than in the others.

As with constant memory, the use of texture memory is thus a compile-time decision whose validity depends on run-time properties of the program input.

Dynamic Parallelism

Another core assumption of our GPU model is that there are only two levels of parallelism: we launch a number of groups, and each group contains a number of threads. Recent GPUs support *dynamic parallelism*, in which new kernels may be launched by any thread in a running kernel, after which the thread may wait for completion of the new kernel instance. This is an instance of the well-known fork-join model of parallelism.

Dynamic parallelism has interesting use cases. One can imagine a search algorithm that starts by performing a coarse analysis of regions of the input data, then launching new kernels for those parts that appear interesting. Very irregular algorithms, such as graph processing, may also be convenient to implement in this fashion.

Unfortunately, dynamic parallelism often suffers from performance problems in practice [DT13; WY14], and fruitful applications are not as prevalent as might have been expected. Efficient usage of dynamic parallelism centers around collecting the desired new kernel launches and merging them together into as few launches as possible, to amortise the overhead. From a functional programming point of view, it seems a better strategy to implement irregular problems via flattening (either manual or automatic), rather than using dynamic parallelism. For these reasons, dynamic parallelism is not part of our GPU model, and is not employed by the Futhark compiler.[1]

Intra-Group Communication

Most GPUs have support for efficient "swizzle" operations for exchanging values between threads in a warp. However, despite the hardware supporting this as a general capability, current programming interfaces provide only specialised variants, for example for summation inside a warp, with each thread providing an integer. In our GPU model, we thus require all cross-thread communication to be done through local memory. This is not a restriction with deep ramifications–should more efficient intra-group communication primitives become available, it would have no great overall impact on our compilation strategy.

4.5 GPU Implementation of Selected Parts of the Array Calculus

The combinators presented in Chapter 3 permit an efficient mapping to GPUs. Indeed, we shall see that "kernel extraction", the process of generating flat parallel GPU kernels from a nested parallel Futhark program, is primarily a process of rewriting the program into a composition of recognisable primitives. This section shows how a subset of these are handled with respect to the GPU model introduced above. In order

[1] Another, more pragmatic, reason is that dynamic parallelism is not supported by the OpenCL implementation provided by NVIDIA.

```
thread(in, out, n):
  gtid <- get_global_id()

  if gtid < n:
    x <- in[gtid]
    y <- f(x)
    out[gtid] <- y
```

Figure 21: A GPU implementation of map $f\ a$. We assume the array a is stored in memory pointed at by the parameter `in`.

to focus on mapping the parallelism and exploiting certain crucial optimisations, we will ignore some technicalities, such as allocating storage for intermediate results inside the functional arguments to the parallel combinators.

The simplest case is implementing map $f\ a$. This can be done with a kernel that conceptually runs one thread for every element in a, which applies f and writes the result. Supose that a contains n elements, and that our desired workgroup size is w. We then use n/w workgroups (rounding up). Since the size of a may not be divisible by the optimal group size for the GPU, the last workgroup might have threads that are idle. The kernel is shown on Figure 21. We assume that f is merely executed sequentially within each thread.

Mapping a reduction such as reduce $\oplus\ 0_\oplus\ a$ is more complicated. Again supposing that a contains n elements, one solution is to launch $n/2$ threads. Each thread reads 2 elements, applies $\oplus$, and writes its result. The process is then recursively applied on the resulting $n/2$ elements until only a single one is left. This implementation is shown on Figure 22.

The reduction thus computed is correct and fully parallel, but it requires manifesting $O(\log n)$ intermediate results in global memory. The time taken to shuffle memory around is likely to significantly dominate the cost of computing $\oplus$. Further, it is awkward if we want to support not just ordinary reductions, but the redomap construct, where reduction is preceded by a mapping. A better solution, one that takes more direct advantage of the GPU architecture, is needed.

An implementation of redomap $\oplus\ f\ 0_\oplus\ a$ is shown on Figure 23. The idea is to divide the n input elements into *chunks*, where each thread processes a chunk of the input sequentially using the f function, followed by all threads within a workgroup collaboratively reducing together, yielding one result per workgroup (which here is written by thread 0 of the group). These per-workgroup results can then be reduced using a simple single-workgroup reduction to a single, final result. The result is that reduction of an arbitrary-sized input array can be done using two GPU kernel invocations.

```
thread(in, out, n):
  gtid <- get_global_id()

  z <- 0⊕

  if gtid < n:
    x <- in[gtid*2]
    y <- in[gtid*2+1]
    z <- x ⊕ y

  out[gtid] <- z
```

Figure 22: A GPU implementation of reduce $\oplus$ $0_\oplus$ a. We assume the array a is stored in memory pointed at by the parameter `in`. The kernel is invoked several times, swapping `in` and `out` for every invocation.

Assume that we are launching k workgroups each of size w, for a total of $k \cdot w$ threads. For simplicity we assume that n is divisible by $k \cdot w$, which means that each thread processes exactly $\frac{n}{k \cdot w}$ elements. This assumption can be removed by having each thread explicitly compute its chunk size. We also assume that w is a power of two, as this simplifies the the intra-group reduction.

Unfortunately, the kernel shown on Figure 23 has a significant problem: the memory accesses performed during the sequential per-chunk processing are non-coalesced. This is because the chunk processed by a given thread is sequential in memory. At an abstract level, we are treating the input as a two-dimensional array of shape $\frac{n}{(k \cdot w)} \times (k \cdot w)$ represented in *row-major order*.

A simple solution is to *stride* the chunks, such that each element within a chunk is separated by $k \cdot w$ elements in the array as a whole. Using the matrix metaphor, this corresponds to representing the input in *column-major order*. This approach is shown in Figure 24. Unfortunately, this solution changes the order of application of $\oplus$, which means that unless $\oplus$ is commutative, the result of the reduction is now wrong. We can solve this by treating the input as a two-dimensional array and *pre-transposing* the input, after which the "wrong" iteration order will actually restore the original order. As shown in Section 4.3.1, transpositions can be implemented efficiently on GPUs, so this extra memory traffic is a small price to pay, if the alternative is non-coalesced access. In practice, most operators we wish to use in reductions are commutative, in which case the pre-transposition is not necessary. A generalisation of this technique is used by the Futhark compiler, as discussed in Section 9.1.

```
local block[]

thread(in, out, k, w, n):
  gtid <- get_global_id()
  ltid <- get_local_id()
  gid <- get_group_id()

  chunk_size <- n/(k*w)

  -- Compute a fold of the chunk.
  i <- 0
  x <- 0⊕
  while i < chunk_size:
    y <- f(in[gtid*chunk_size + i])
    x <- x ⊕ y
    i <- i + i

  block[ltid] <- x
  barrier()

  i <- w/2
  while i > 0:
    if ltid < i:
      x <- block[ltid*2]
      y <- block[ltid*2+1]
      z <- x ⊕ y
    barrier()
    if ltid < i:
      block[ltid] <- z
    barrier()
    i <- i / 2

  if ltid == 0:
    out[gid] <- block[0]
```

Figure 23: A GPU implementation of redomap $\oplus$ $0_\oplus$ a. We assume the array a is stored in memory pointed at by the parameter `in`. The result is one partial result per workgroup, which will have to be processed by a final (much smaller) reduction.

```
local block[]

thread(in, out, k, w, n):
  ...

  -- Compute a fold of the chunk.
  i <- 0
  x <- 0⊕
  while i < chunk_size:
    y <- f(in[gtid + i*(w*n)])
    x <- x ⊕ y
    i <- i + i

  ...
```

Figure 24: A rewriting of the per-chunk folding of Figure 23 so that memory accesses are coalesced. This only works if the reduction is commutative.

Part II

An Optimising Compiler

Chapter 5

Overview and Uniqueness Types

This chapter contains three primary elements: First, an overview of the compiler design, with an emphasis on how the program is transformed as it proceeds through the pipeline (Section 5.1). Second, a simplified Futhark core language that will be used in the following chapters to describe the operations performed by the Futhark compiler (Section 5.2). Third, a formalisation of the uniqueness type system expressed on the core language (Section 5.3).

The simplification engine encompasses well established optimisations such as inlining, copy propagation, constant folding, common-subexpression elimination, dead-code elimination, and hoisting of invariant terms out of recurrences in loops and SOACs. The simplification rules are critical for optimising the result of higher-level optimisations, such as producer-consumer fusion [Hen14; HO13], hoisting, and size analysis, which is the subject of Chapter 6. For example, size analysis is implemented via high-level transformation rules that only guarantee that the asymptotic complexity (in number of operations) of the original program is preserved, but rely on the simplification engine to reduce the overhead to be negligible in the common case.

5.1 Compiler Design

The Futhark compiler is implemented in Haskell, and is designed as a conventional syntax-directed translator. The compiler is structured as a sequence of *passes*, each of which can be seen as functions that accept a program as input, and produce a program as output. Some passes perform rewrites of a program in some representation, while other passes lower the program to a more specialised or lower-level representation.

The compiler contains two different compiler pipelines: one for generating a sequential program, and one for generating a program with parallel code. Parallelism in the generated code is expressed as OpenCL [SGS10] kernels. While OpenCL is a hardware-agnostic API, and we generate portable OpenCL code, many of the optimisations we perform make assumptions that are specific to GPUs. We might

not achieve desired performance if we executed the generated OpenCL code on, for example, FPGAs or other such exotic platforms. We have, however, noted decent performance on multicore CPUs, but this is not a direction we shall investigate in this thesis.

The initial passes of the sequential and parallel pipelines are identical, but they differ after a certain point. In this section, and the thesis as a whole, we will focus on the parallel compilation pipeline, as it is by far the more complicated of the two. The intermediate representation used by Futhark is typed, and to ensure robustness, full type-checking (including verifying safety of in-place updates) is performed between all compiler passes. The type checking rules become more lenient as the representation is gradually lowered, and the later stages perform only cursory checking. This type-checking carries a significant compilation overhead (approximately 20%), so we expect to disable it for production releases of the compiler.

The pipeline proceeds with the following passes in order:

Internalisation and Desugaring: The source Futhark language is translated into the core language (Section 5.2). In particular, modules and polymorphic functions are removed through specialisation (outside the scope of this thesis), all tuples are flattened, and arrays of tuples are transformed into tuples of arrays, using the technique shown on Figure 15. This is also the pass where where size inference is performed (Chapter 6).

Inlining and Dead Function Removal: All user-defined functions are inlined. As the Futhark language presently does not support recursive functions, this is always possible. This may result in significant growth in code size, and so future work should investigate more conservative approaches. Inlining is critical to making subsequent optimisations possible, however. On GPUs, our main target platform, all functions inside kernels are automatically inlined in any case, so we do not lose much by inlining at the Futhark-level as well.

SOAC Fusion: SOACs in a producer-consumer relationship, or simply adjacent and looping over arrays of identical size, are fused (Chapter 7).

Kernel Extraction: A moderate flattening algorithm is applied to the program, which transforms nests of SOACS into flat-parallel kernels (Chapter 8). The kernels produced here are not yet low-level GPU kernels, but a higher-level representation that still abstracts many details, in particular issues of memory allocation. Nested parallelism is gone after this pass.

Coalescing Fixes: The kernels generated in the previous pass are inspected, and non-coalesced array access patterns are resolved by semantically transposing the arrays in question (Section 9.1).

Loop Tiling: Loop tiling is performed inside the generated kernels (Section 9.2). This is the last pass we discuss in detail in this thesis—the remainder are concerned with various technical minutiae, which, while extremely time-consuming and challenging to implement, are of lesser scientific importance.

Memory Allocation: Explicit allocations are inserted in the program, in a scheme similar to region inference [Tof+04]. Every array becomes associated with a "memory block", the allocation of which can be hoisted out of the recurrence or branch where the array itself is constructed. To handle cases where pre-allocation is not possible (such as a loop where the array size changes for every iteration), the existential-types mechanism also employed for size inference (Chapter 6) is used. The ability of the compiler to manipulate memory allocations as program constructs is important. We use this to arrange the program such that all memory is pre-allocated before entering GPU kernels, as allocation is not possible inside of these.

Double Buffering: We perform double buffering to enable the hosting of memory allocations out of loops (in particular loops that are inside parallel kernels). For technical reasons, we currently do not perform double buffering in the usual way, with pointer swapping, but instead insert extra copies for every iteration of the loop. In Section 10.1.1 we shall see an example where this significantly impacts our performance.

Memory Expansion: It is not safe to naively hoist allocations out of parallel kernels. When hoisting such allocations, we first have to perform *expansion*, such that one per-thread private allocation of b bytes becomes one shared allocation of $n \times b$ bytes, where n is the number of threads. This requires all allocations to be at the top level of kernels, and for their size to be expressible in terms of (or at least bounded by) variables whose value is known before the kernel invocation.

Imperative IR Code Generation: The flat-parallel Futhark program with explicit allocations is transformed into a low-level imperative intermediate representation (IR). This IR is similar to C, but significantly simpler, and much more explicit. No optimisations are performed on this representation. The IR has special constructs for representing GPU kernels, whose bodies are also expressed with the imperative IR. This pass fails if any allocations remain inside parallel kernels.

Final Code Generation: The imperative IR is transformed into host-level (CPU) code and actual OpenCL kernels. The host-level code is either C or Python, with calls to the OpenCL library, while OpenCL kernels are in OpenCL C (a restricted dialect of C). We write the result to one or more files, and, depending on the compilation options, may invoke a C compiler to generate an executable.

5.1.1 Simplification Passes

The preceding list of passes is incomplete. Between almost every pass, we perform a range of simplifications and conventional optimisations. These are too numerous, and too individually uninteresting, to be discussed in detail, but a few deserve mention:

Copy propagation, constant folding, and CSE: The usual bevy of of scalar optimisations is performed, with range analysis and propagation performed to support the elimination of (some) bounds- and size checks.

Aggressive hoisting: The Futhark compiler aggressively hoists expressions out of loops and branches, where possible and safe. Care is taken to avoid hoisting potentially expensive constructs out of branches.

Dead code removal: We remove not merely expressions whose results are never used, but also rewrite expressions where only part of the result is used. Consider for example an `if` that returns a pair, where only one component of the pair is subsequently used.

Removal of single-iteration loops and SOACs: Any loop with a single iteration, or a SOAC whose input array has a constant size of 1, can be trivially substituted with its body. This is used to enable efficient sequentialisation (see Section 7.3.2 for an example).

Un-existentialisation: As we shall see in Chapter 6, Futhark uses an existential type system with size-dependent types. Simplification rules are used for making existential sizes concrete where possible.

Most of the simplifications are performed by a general simplification engine, which traverses the program and applies a collection of rewrite rules to every expression. The process is repeated until a fixed point is reached. This approach is fairly similar to the one used by the Glasgow Haskell Compiler (GHC) [JTH01]. The main difference is that in GHC the rules are expressed as user-provided equalities, while the simplification rules in Futhark are hardwired into the compiler in the form of arbitrary Haskell functions. This allows the Futhark rewrite rules to be significantly more powerful, as they can inspect the surrounding environment, but they are not user-extensible. As an example, Figure 25 defines a rule that removes results of a `map` that are not used subsequently. This rule depends on information about what happens *after* the expression, and is thus applied during a bottom-up traversal. This figure constitutes the sole exception to our general policy of not showing compiler implementation code in this thesis.

```
removeDeadMapping :: (MonadBinder m,
                      Op (Lore m) ~ SOAC (Lore m))
                     => BottomUpRule m
removeDeadMapping (_, used)
                  (Let pat _ (Op (Map width fun arrs))) =
  let ses = bodyResult $ lambdaBody fun
      isUsed (bindee, _, _) =
        (`UT.used` used) $ patElemName bindee
      (pat',ses', ts') =
        unzip3 $ filter isUsed $
        zip3 (patternElements pat) ses $ lambdaReturnType fun
      fun' = fun { lambdaBody =
                     (lambdaBody fun) { bodyResult = ses' }
                 , lambdaReturnType = ts'
                 }
  in if pat /= Pattern [] pat'
     then letBind_ (Pattern [] pat') $
            Op $ Map width fun' arrs
     else cannotSimplify
removeDeadMapping _ _ = cannotSimplify
```

Figure 25: An example of a simplification rule from the Futhark compiler (slightly reformatted for presentation purposes). This rule removes the computation of those results of a **map** that are never subsequently used.

5.2 Abstract Syntax of the Core IR

This section describes the abstract syntax of a simplified form of the Futhark intermediate representation (IR). We omit some book-keeping information that is convenient when implementing the compiler, but would merely clutter the exposition here.

5.2.1 Notation for sequences

Whenever z is an object of some kind, we write $\overline{z}$ to range over sequences of objects of this kind. When we want to be explicit about the size of a sequence $\overline{z} = z_0, \cdots, z_{(n-1)}$, we often write it on the form $\overline{z}^{(n)}$ and we write $z, \overline{z}$ to denote the sequence $z, z_0, \cdots, z_{(n-1)}$. Depending on context, the elements of the sequence or may have separators, or be merely juxtaposed. For example, we may the same notation to shorten a function application

$$f\ \overline{v}^{(n)} \equiv f\ v_1\ \cdots\ v_n$$

or a tuple

$$(\overline{v}^{(n)}) \equiv (v_1, \ldots, v_n)$$

or a function type

$$\overline{\tau}^{(n)} \to \tau_{n+1} \equiv \tau_1 \to \cdots \to \tau_n \to \tau_{n+1}.$$

It is always clear from the context which separator (if any) is intended. On occasion we use more complicated sequences, where not all terms under the bar are variant. In this case, the variant term is subscripted with i. For example,

$$(\overline{[d]v_i}^{(n)}) = ([d]v_1, \ldots, [d]v_n)$$

and

$$(\overline{[d_i]v_i}^{(n)}) = ([d_1]v_1, \ldots, [d_n]v_n)$$

5.2.2 AST definition

Figure 26 shows the (simplified) core IR of Futhark, which will be used for all compiler chapters of this thesis. The syntax is intentionally similar to the source language. Notable details include:

- Variable names are ranged over by d, x, y, and z, and we use f to range over function names.
- A variable may have a scalar type (i.e., `bool`, `i32`, `f64`), or a multidimensional (regular) array type, such as `[][]f64`, which is the type of a matrix in which all rows have the same size.
- Named and unnamed functions use a syntax similar to the source language, but unnamed functions may appear only as immediate arguments to SOACS, such as **`map`**, **`reduce`**, **`filter`**, and so on. Moreover, named functions may only be defined at top-level. Recursion is not permitted.

When discussing properties of the language, we will often assume that expressions (e) in the intermediate language are in A-normal form [SF92]. That is, all subexpressions are variable names (except for the special cases **`loop`**, **`if`**, and **`let`**). However, for readability, we shall often stray from strict A-normal form when showing examples. Tuple-typed variables do not exist in the IR, except as a syntactical construct for some operators. A let-bound pattern (p) consists of one or more variables and associated types that bind the result of an expression (e). This is intuitively equivalent to a tuple pattern in the source language. Types are syntactically differentiated into two sorts: τ, without uniqueness attributes, and $\hat{\tau}$, which permit an optional uniqueness attribute (an asterisk). Uniqueness attributes are only used when specifying the parameter and return types of top-level functions and **`for`**-loops—patterns in **`let`** bindings and anonymous functions possess no uniqueness attributes.

f	::=	**id**	(Function names)
d, x, y, z	::=	**id**	(Variable names)
c	::=	**const**	(Constant value)
t	::=	`bool` | `i32` | `f32` | ...	(Built-in types)
τ	::=	`t` | `[]`τ	(Scalar/array type)
$\hat{\tau}$	::=	τ | $\star\tau$	(Nonunique/Unique type)
ρ	::=	$(\tau_1, \ldots, \tau_n)$	(Tuple types)
$\hat{\rho}$	::=	$(\hat{\tau}_1, \ldots, \hat{\tau}_n)$	(Nonunique/Unique tuple types)
ϕ	::=	$\rho_1 \to \cdots \to \rho_n$	(Function type)
p	::=	$(x_1 : \tau_1)\ \ldots\ (x_n : \tau_n)$	(**let** or λ pattern)
$\hat{p}$	::=	$(x_1 : \hat{\tau}_1)\ \ldots\ (x_n : \hat{\tau}_n)$	(Function pattern)
fun	::=	**let** $f\ \hat{p}$ **:** $\hat{\tau} = e$	(Named function)
P	::=	ϵ | $fun\ P$	(Program)
e	::=	$(x_1, \ldots, x_n)$	(n-tuple)
	|	**let** $p = e$ **in** b	(Let binding)
	|	k	(Constant)
	|	x	(Variable)
	|	$e \odot e$	(Scalar binary operator)
	|	**if** e **then** e **else** e	(Branch)
	|	$x\,[x_1, \ldots, x_n]$	(Array indexing)
	|	x **with** $[e_1, \ldots, e_n] \leftarrow e$	(In-place update)
	|	**loop** $(\hat{p}) = (x_1, \ldots, x_n)$ **for** $x < x$ **do** e	(Loop)
	|	$f\ x_1\ \ldots\ x_n$	(Function call)
	|	$op\ a_1\ \ldots\ a_n$	(Operator call)
a	::=	x	(Simple argument)
	|	$(x_1, \ldots, x_n)$	(Tuple argument)
	|	$(\lambda p : \rho \to e)$	(Function argument)

Figure 26: Grammar for the core Futhark IR. Some compiler stages may impose additional constraints on the structure of ASTs, in particular by requiring size annotations to be present, or banning certain operations. The definition of *op* in particular may differ between stages. Some constructs, such as **while** loops, have been elided for simplicity. The elided constructs would have no influence on the development of the thesis.

Syntactic Conveniences

We permit the omission of of **in** immediately preceding **let**. This allows us to write

```
let x = 2
let y = 3
in x + y
```

instead of

```
let x = 2 in
let y = 3 in
x + y
```

which quickly adds up for larger expressions. This convenience is also present in the Futhark source language. For patterns in **let**-bindings, we will also write

$$(x_1 : \tau_1, \ldots, x_n : \tau_n)$$

instead of

$$(x_1 : \tau_1) \ldots (x_n : \tau_n)$$

as strictly required by the syntax, in order to match the source language. For brevity, we will also sometimes elide some type annotations. In such cases we may also elide the parentheses when only a single name is bound by the pattern. We will often stray from the strict syntax for lambdas, and write for example `(+)` for `(λx y→x+y)`, or `(+2)` instead of `(λx→x+2)`.

On occasion, we will need to bind variables whose names we do not care about. In these cases we will use an underscore to indicate that the name is irrelevant. For example:

```
map (λ_ y _ → y + 2) xs ys zs
```

5.2.3 Array Operators

An *array operator* (*op*) is one of several constructs that operate on arrays. These include SOACs, but also such operations as **iota** and **reshape**. An array operator can be applied to multiple arguments (a), where an argument can be a variable in scope, but may also be a tuple of variables, or an anonymous function.

For giving concise types to operators, we use a notion of *extended types* that supports polymorphism in types and for which arguments to functions may themselves be functions:

$$\begin{array}{lcl} \underline{\tau} & ::= & \alpha \mid (\tau_1, \ldots, \tau_n) \mid \underline{\tau}_1 \to \underline{\tau}_2 \\ \underline{\phi} & ::= & \forall \overline{\alpha}.\underline{\tau} \end{array}$$

op $a_1 \ \ldots \ a_n$	Description
size $c \ x$	Returns the size of dim c of x, where c must be a non-negative integer literal.
iota x	Returns the vector $[0, \ldots, x-1]$.
replicate $y \ x$	Returns an array of rank one higher than x's rank, containing an y-times replication of x.
reshape $(y_1, \ldots, y_n) \ x$	Return an array of shape $y_1 \times \ldots \times y_n$ containing the row-major elements of x. It is a dynamic error if x does not contain exactly $y_1 \times \ldots \times y_n$ elements.
rearrange $(c_1, \ldots, c_n) \ x$	Rearrange the order of dimensions of the n-dimensional array x. The reordering $c_1, \ldots, c_n$ must be a permutation of 1...n.

Figure 27: Description of array operators. SOACs are described on Figure 28

op $\overline{a}^{(l)}$	Description
map $\lambda \ \overline{x}^{(n)}$	Apply the n-ary function λ simultaneously to consecutive elements of $x_1 \ldots x_n$, producing an array of the results. If λ returns k values, k arrays are returned.
scatter $x \ y \ z$	For all i, write $z[i]$ to $x[y[j]]$. Consumes x and semantically returns a new array. The result is unspecified if different values are written to the same position. Out-of-bound writes have no effect.
reduce $\lambda \ (\overline{y}^{(n)}) \ \overline{x}^{(n)}$	Perform a reduction of arrays $x_1, \ldots, x_n$ via the $2n$-ary function λ, producing n values. The values $y_1, \ldots, y_n$ constitute a neutral argument for λ.
scan $\lambda \ (\overline{y}^{(n)}) \ \overline{x}^{(n)}$	Perform an inclusive prefix scan of arrays $x_1, \ldots, x_n$ via the $2n$-ary function λ, producing n values. The values $y_1, \ldots, y_n$ constitute a neutral argument for λ.
filter $\lambda \ \overline{x}^{(n)}$	Produce n arrays, containing only those elements with index j for which $\lambda \ \overline{x_i[j]}^{(n)}$ is true.
stream_map $\lambda \ \overline{x}^{(n)}$	See Section 2.5.2.
stream_red $\lambda_c \ \lambda_f \ \overline{x}^{(n)}$	See Section 2.5.2.

Figure 28: Description of SOAC operators.

op		TySch(*op*)
size	:	$\forall\alpha.\texttt{i32} \to \alpha \to \texttt{i32}$
iota	:	$\texttt{i32} \to []\texttt{i32}$
replicate	:	$\forall\alpha.\texttt{i32} \to \alpha \to []\alpha$
reshape	:	$\forall\alpha.(\texttt{i32}_1, \ldots, \texttt{i32}_n) \to []_1 \cdots []_m\alpha \to []_1 \cdots []_n\alpha$
rearrange $(\overline{c}^{(n)})$	:	$\forall\alpha.[]_1 \cdots []_n\alpha \to []_1 \cdots []_n\alpha$
map	:	$\forall\overline{\alpha}^{(n)}\overline{\beta}^{(m)}.(\overline{\alpha}^{(n)} \to (\overline{\beta}^{(m)})) \to \overline{[]\alpha}^{(n)} \to (\overline{\beta}^{(m)})$
scatter	:	$\forall\overline{\alpha}^{(n)}.(\overline{[]\alpha_i}^{(n)}) \to (\overline{[]\texttt{i32}}^{(n)}) \to (\overline{[]\alpha}^{(n)}) \to (\overline{[]\beta}^{(n)})$
reduce	:	$\forall\overline{\alpha}^{(n)}.(\overline{\alpha}^{(n)} \to \overline{\alpha}^{(n)} \to (\overline{\alpha}^{(n)})) \to (\overline{\alpha}^{(n)}) \to \overline{[]\alpha}^{(n)} \to (\overline{\alpha}^{(n)})$
scan	:	$\forall\overline{\alpha}^{(n)}.(\overline{\alpha}^{(n)} \to \overline{\alpha}^{(n)} \to (\overline{\alpha}^{(n)})) \to (\overline{\alpha}^{(n)}) \to \overline{[]\alpha}^{(n)} \to (\overline{[]\alpha}^{(n)})$
filter	:	$\forall\overline{\alpha}^{(n)}.(\overline{\alpha}^{(n)} \to \texttt{bool}) \to \overline{[]\alpha}^{(n)} \to (\overline{[]\alpha}^{(n)})$
stream_map	:	$\forall\overline{\alpha}^{(n)}\ \overline{\beta}^{(m)}.(\overline{[]\alpha_i}^{(n)} \to (\overline{[]\beta_i}^{(m)})) \to \overline{[]\alpha}^{(n)} \to (\overline{[]\beta}^{(m)})$
stream_red	:	$\forall\overline{\alpha}^{(n)}\ \overline{\beta}^{(m)}.(\overline{\beta}^{(m)} \to \overline{\beta}^{(m)} \to (\overline{\beta}^{(m)})) \to (\overline{[]\alpha_i}^{(n)} \to (\overline{\beta_i}^{(m)})) \to \overline{[]\alpha}^{(n)} \to (\overline{\beta}^{(m)})$

Figure 29: Size-agnostic type schemes for operators, including various SOACs.

Extended types ($\underline{\tau}$) and extended type schemes ($\underline{\sigma}$) are used only for the treatment of operators and we shall be implicit about converting types and type schemes to and from their extended counterparts. We treat the single-element tuple (τ) as equivalent to τ. A *substitution* (S) is a mapping from type variables (and, later, term variables) to extended types. We write substitutions $\langle x \mapsto y \rangle$. Applying a substitution S to some object B, written $S(B)$, has the effect of simultaneously applying S to variables in B (being the identity outside its domain). An extended type $\underline{\tau}'$ is *an instance of* an extended type scheme $\underline{\phi} = \forall\vec{\alpha}.\underline{\tau}$, written $\underline{\phi} \geq \underline{\tau}'$, if there exists a substitution S such that $S(\underline{\tau}) = \underline{\tau}'$.

Type schemes for operators, including a representative subset of the SOAC operators, are given in Figure 29, and an informal description is given in Figures 27 and 28. The SOACs of the intermediate language (such as **map**) are a tuple-of-array versions of the user language SOACs. The SOACs of the intermediate language receive an arbitrary number of array arguments and produce a tuple of arrays as their result. The semantics of a SOAC operator can be intuitively understood as a composition between **unzip**, the user-language SOAC (such as **map**), and **zip**, where the unnamed function is suitably modified to work with the flat sequence of array arguments.

5.2.4 Typing Rules

In the type rules, we permit the implicit transformation of uniqueness types and patterns $\hat{\tau}/\hat{\rho}/\hat{p}$ to their corresponding $\tau/\rho/p$, where uniqueness attributes have simply been removed. This simplifies the type rules, which can then be stated separate from

the rules for checking uniqueness properties.

Type environments (Γ) are finite maps from program variables to types or function types. Looking up the type of x in Γ is written $\Gamma(x)$. When Γ is a type environment and p is a pattern $\tau_1\ x_1, \cdots, \tau_n\ x_n$, we write Γ, p to denote the typing environment $\Gamma, x_1 : \tau_1, \cdots, x_n : \tau_n$.

Typing rules for the source language are given in Figures 30 and 31. The rules allow inferences among sentences of the following forms:

$\vdash_{\mathbf{p}} p : \rho$**,** read "the pattern p is matched by expressions of tuple type ρ."

$\vdash_{\hat{\mathbf{p}}} \hat{p} : \hat{\rho}$**,** read "the pattern with uniqueness $\hat{p}$ is matched by expressions of tuple type ρ."

$\Gamma \vdash_{\mathbf{a}} a : \tau/\phi$**,** read "under the assumptions Γ, the operator argument a has type τ or function type ϕ."

$\Gamma \vdash e : \rho$**,** read "under the assumptions Γ, the expression e has tuple type ρ."

$\Gamma \vdash_{\mathbf{P}} P : \rho$**,** read "under the assumptions Γ, the program P is well-typed."

Patterns $\boxed{\vdash_{\mathrm{p}} p : \rho}$

$$\frac{}{\vdash_{\mathrm{p}} (x_1 : \tau_1, \cdots, x_n : \tau_n) : (\tau_1, \cdots, \tau_n)} \tag{5.1}$$

Operator arguments $\boxed{\vdash_{\mathrm{a}} a : \rho/\phi}$

$$\frac{}{\Gamma \vdash_{\mathrm{a}} x : \Gamma(x)} \tag{5.2}$$

$$\frac{\vdash_{\mathrm{p}} p : \rho \quad \Gamma, p \vdash e : \rho'}{\Gamma \vdash_{\mathrm{a}} (\lambda p : \ \rho \ \rightarrow e) : \rho \rightarrow \rho'} \tag{5.3}$$

$$\frac{}{\Gamma \vdash_{\mathrm{a}} (x_1, \ldots, x_n) : (\Gamma(x_1), \ldots, \Gamma(x_n))} \tag{5.4}$$

Parameters $\boxed{\vdash_{\hat{\mathrm{p}}} \hat{p} : \hat{\rho}}$

$$\frac{}{\vdash_{\hat{\mathrm{p}}} (x_1 : \hat{\tau}_1, \cdots, x_n : \hat{\tau}_n) : (\hat{\tau}_1, \cdots, \hat{\tau}_n)} \tag{5.5}$$

Programs $\boxed{\Gamma \vdash_{\mathrm{P}} P}$

$$\frac{\begin{array}{c}\vdash_{\hat{\mathrm{p}}} \hat{p} : (\tau_1, \ldots, \tau_n) \quad \Gamma, \hat{p} \vdash e : \rho \\ \Gamma(f) = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \hat{\rho} \quad \Gamma \vdash_{\mathrm{P}} P\end{array}}{\Gamma \vdash_{\mathrm{P}} \mathtt{let} \ f \ \hat{p} \ : \ \hat{\rho} \ = \ e \ P} \tag{5.6}$$

$$\frac{}{\Gamma \vdash_{\mathrm{P}} \epsilon} \tag{5.7}$$

Figure 30: Typing rules for Futhark's core IR, excluding expressions, which are shown on Figure 31.

Expressions $\boxed{\Gamma \vdash e : \rho}$

$$\frac{\vdash_{\mathrm{p}} p : \rho \quad \Gamma \vdash e_1 : \rho \quad \Gamma, p \vdash e_2 : \rho'}{\Gamma \vdash \mathbf{let}\ p = e_1\ \mathbf{in}\ e_2 : \rho'} \tag{5.8}$$

$$\frac{}{\Gamma \vdash (x_1, \cdots, x_n) : (\Gamma(x_1), \cdots, \Gamma(x_n))} \tag{5.9}$$

$$\frac{\Gamma(x) = []\tau \quad \Gamma(s) = \texttt{i32}}{\Gamma \vdash x[s] : \tau} \tag{5.10}$$

$$\frac{\mathrm{ConstType}(ct) = \tau}{\Gamma \vdash c : \tau} \tag{5.11}$$

$$\frac{\Gamma \vdash_{\mathrm{a}} a_i : \underline{\tau}_i \quad i \in \{1, \ldots, n\} \quad \mathrm{TySch}(op) \geq (\overline{\underline{\tau}}^{(n)}) \to \rho}{\Gamma \vdash op\ \overline{a}^{(n)} : \rho} \tag{5.12}$$

$$\frac{\Gamma(x_i) = \tau_i \quad i \in \{1, \ldots, n\} \quad \mathrm{lookup}_{\mathrm{fun}}(f) = \tau_1 \to \cdots \to \tau_n \to \rho}{\Gamma \vdash f\ x_1\ \ldots\ x_n : \rho} \tag{5.13}$$

$$\frac{\Gamma(s) = \texttt{bool} \quad \Gamma \vdash e_1 : \rho \quad \Gamma \vdash e_2 : \rho}{\Gamma \vdash \mathbf{if}\ s\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \rho} \tag{5.14}$$

$$\frac{\vdash_{\mathrm{p}} \hat{p} : \rho \quad \Gamma \vdash e_1 : \rho \quad \Gamma \vdash e_3 : \rho \quad \Gamma \vdash e_2 : \texttt{i32}}{\Gamma \vdash \mathbf{loop}\ \hat{p} = e_1\ \mathbf{for}\ x < e_2\ \mathbf{do}\ e_3 : \rho} \tag{5.15}$$

Figure 31: Typing rules for expressions in Futhark's core IR. The defition of TySch is found on Figure 29.

5.3 Checking Uniqueness Types

An in-place update `a` **`with`** `[i]` $\leftarrow$ `v` is safe if and only if:

1. The array `a` (including aliases) does not occur on any execution path following the in-place update.
2. `v` does not alias `a`.

To check for safety statically, we require two things: aliasing information for all variables in the program (Section 5.3.1), and a way to check for how and when variables are used (Section 5.3.2). In the formalisation that follows, alias analysis is separate from safety checking. In an implementation, the two can be done in a single pass.

5.3.1 Alias Analysis

We perform alias analysis on a program that we assume to be otherwise type-correct. Our presentation uses an inference rule-based approach in which the central judgment takes the form $\Sigma \vdash e \Rightarrow \langle \sigma_1, \ldots, \sigma_n \rangle$, which asserts that, within the context Σ, the expression e produces n values, where value number i has the *alias set* σ_i. An alias set is a subset of the variable names in scope, and indicates which variables an array value (or variable) may be aliased with. The context Σ maps variables in scope to their aliasing sets.

The aliasing rules are listed in Figures 32 and 33. The ALIAS-LETPAT-rule ensures that the aliases of a variable include the variable itself—that is, $v \in \Sigma(v)$. This property is maintained by all binding constructs. Alias sets for values produced by SOACs such as **`map`** are empty. We can imagine the arrays produced as *fresh*, although the compiler is of course free to reuse memory if it can do so safely. The ALIAS-INDEXARRAY rule tells us that a scalar read from an array does not alias its origin array, but ALIAS-SLICEARRAY dictates that an array slice does. This fits the our intuition of how an implementation might implement these cases - scalars are read into registers, where array slicing is just offsetting a pointer.

The most interesting aliasing rules are the ones for function calls (ALIAS-APPLY-NONUNIQUE and ALIAS-APPLY-UNIQUE). Since our alias analysis is intra-procedural, we are forced to be conservative. There are two rules, corresponding to functions returning unique and non-unique arrays, respectively. When the result is unique the alias set is empty, otherwise the result conservatively aliases all non-unique parameters.

5.3.2 In-Place Update Checking

In our implementation, alias computation and in-place update checking is performed at the same time, but is split here for expository purposes. In the following, let

Expression aliases $\boxed{\Sigma \vdash e \Rightarrow \langle \sigma_1, \ldots, \sigma_n \rangle}$

$$\frac{}{\Sigma \vdash (x_1, \ldots, x_n) \Rightarrow \langle \{x_1\} \cup \Sigma(x_1), \ldots, \{x_n\} \cup \Sigma(x_n) \rangle} \quad \text{(ALIAS-TUPLE)}$$

$$\frac{\begin{array}{c}\Sigma \vdash e_1 \Rightarrow \langle \overline{\sigma}^{(n)} \rangle \\ \Sigma, x_i \mapsto \sigma_i \vdash e_2 \Rightarrow \langle \overline{\sigma'}^{(n)} \rangle\end{array}}{\Sigma \vdash \texttt{let } \overline{(x : \tau)}^{(n)} = e_1 \texttt{ in } e_2 \Rightarrow \langle \overline{\sigma'_i \cup \{x_i\}}^{(n)} \rangle} \quad \text{(ALIAS-LETPAT)}$$

$$\frac{}{\Sigma \vdash k \Rightarrow \langle \emptyset \rangle} \quad \text{(ALIAS-CONST)}$$

$$\frac{}{\Sigma \vdash x \Rightarrow \Sigma(x)} \quad \text{(ALIAS-VAR)}$$

$$\frac{}{\Sigma \vdash x \odot y \Rightarrow \langle \emptyset \rangle} \quad \text{(ALIAS-SCALARBINOP)}$$

$$\frac{\Sigma \vdash e_2 \Rightarrow \langle s_1^2, \ldots, s_n^2 \rangle \quad \Sigma \vdash e_3 \Rightarrow \langle s_1^3, \ldots, s_n^3 \rangle}{\Sigma \vdash \texttt{if } x_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow \langle s_1^2 \cup s_1^3, \ldots, s_n^2 \cup s_n^3 \rangle} \quad \text{(ALIAS-IF)}$$

$$\frac{x \text{ is of rank } n}{\Sigma \vdash x[\overline{x}^{(n)}] \Rightarrow \langle \emptyset \rangle} \quad \text{(ALIAS-INDEXARRAY)}$$

$$\frac{x \text{ is of rank} > n}{\Sigma \vdash x[\overline{x}^{(n)}] \Rightarrow \langle \{x\} \cup \Sigma(x) \rangle} \quad \text{(ALIAS-SLICEARRAY)}$$

$$\frac{}{\Sigma \vdash x_a \texttt{ with } [\overline{e}^{(n)}] \leftarrow e_x \Rightarrow \langle \Sigma(x_a) \rangle} \quad \text{(ALIAS-UPDATE)}$$

Figure 32: Aliasing rules for simple expressions. For simplicity, we treat only the single-parameter case for loops and function calls.

aliases(v) be the alias set of the variable v, which we assume has been computed as in the previous section. We denote by O the set of the variables *observed* (used) in some expression e, and by C the set of variables *consumed* through function calls and in-place updates. Together, the pair $\langle C, O \rangle$ is called an *occurrence trace*.

Figure 34 defines a *sequencing* judgment between two occurrence traces, which takes the form

$$\langle C_1, O_1 \rangle \gg \langle C_2, O_2 \rangle : \langle C_3, O_3 \rangle$$

and which can be derived if and only if it is acceptable for $\langle C_1, O_1 \rangle$ to happen first, then $\langle C_2, O_2 \rangle$, giving the combined occurrence trace $\langle C_3, O_3 \rangle$. We formulate this as a judgment because sequencing is sometimes not derivable—for example in the case where an array is used after it has been consumed. The judgment is defined by a

Expression aliases (continued) $\boxed{\Sigma \vdash e \Rightarrow \langle \sigma_1, \ldots, \sigma_n \rangle}$

$$\frac{\tau \text{ is not of form } *\tau' \quad \Sigma \vdash x_1 \Rightarrow \langle \sigma \rangle \quad \Sigma, x_1 \mapsto \sigma \cup \{x_1\} \vdash e \Rightarrow \langle \sigma' \rangle}{\Sigma \vdash \textbf{loop}\ (x_1 : \tau)\ =\ y_2\ \textbf{for}\ z_1\ <\ z_2\ \textbf{do}\ e \Rightarrow \langle \sigma' \setminus \{x_1\} \rangle} \text{(ALIAS-DOLOOP-NONUNIQUE)}$$

$$\frac{}{\Sigma \vdash \textbf{loop}\ (x_1 : *\tau)\ =\ y_2\ \textbf{for}\ z_1\ <\ z_2\ \textbf{do}\ e \Rightarrow \langle \emptyset \rangle} \text{(ALIAS-DOLOOP-UNIQUE)}$$

$$\frac{\text{lookup}_{\text{fun}}(f) = \hat{\tau}_1 \to \cdots \to \hat{\tau}_n \to \tau \quad \Sigma \vdash x_i \Rightarrow \langle \sigma_i \rangle \quad \sigma = \bigcup_i (\sigma_i \quad \text{if } \tau_i \text{ is not of form } *\tau')}{\Sigma \vdash f\ x_1\ \ldots\ x_n \Rightarrow \langle \sigma \rangle} \text{(ALIAS-APPLY-NONUNIQUE)}$$

$$\frac{\text{lookup}_{\text{fun}}(f) = \hat{\tau}_1 \to \cdots \to \hat{\tau}_n \to *\tau}{\Sigma \vdash f\ x_1\ \ldots\ x_n \Rightarrow \langle \emptyset \rangle} \text{(ALIAS-APPLY-UNIQUE)}$$

$$\frac{}{\Sigma \vdash \textbf{size}\ c\ x \Rightarrow \langle \emptyset \rangle} \text{(ALIAS-SIZE)}$$

$$\frac{}{\Sigma \vdash \textbf{iota}\ x \Rightarrow \langle \emptyset \rangle} \text{(ALIAS-IOTA)}$$

$$\frac{}{\Sigma \vdash \textbf{replicate}\ x\ y \Rightarrow \langle \emptyset \rangle} \text{(ALIAS-REPLICATE)}$$

$$\frac{\Sigma \vdash y \Rightarrow \sigma}{\Sigma \vdash \textbf{reshape}\ (x_1, \ldots, x_n)\ y \Rightarrow \langle \sigma \rangle} \text{(ALIAS-RESHAPE)}$$

$$\frac{\Sigma \vdash y \Rightarrow \sigma}{\Sigma \vdash \textbf{rearrange}\ (c_i, \ldots, c_n)\ y \Rightarrow \langle \sigma \rangle} \text{(ALIAS-REARRANGE)}$$

$$\frac{}{\Sigma \vdash \textbf{map}\ (\lambda p\textbf{:}\ (\overline{\tau}^{(m)}) \to e)\ \overline{x}^{(n)} \Rightarrow \langle \overline{\emptyset}^{(m)} \rangle} \text{(ALIAS-MAP)}$$

$$\frac{}{\Sigma \vdash \textbf{scatter}\ x\ y\ z \Rightarrow \langle \{x\} \cup \Sigma(x) \rangle} \text{(ALIAS-SCATTER)}$$

$$\frac{}{\Sigma \vdash \textbf{reduce}\ a\ (\overline{x}^{(n)})\ \overline{x}^{(n)} \Rightarrow \langle \overline{\emptyset}^{(n)} \rangle} \text{(ALIAS-REDUCE)}$$

$$\frac{}{\Sigma \vdash \textbf{scan}\ a\ (\overline{x}^{(n)})\ \overline{x}^{(n)} \Rightarrow \langle \overline{\emptyset}^{(n)} \rangle} \text{(ALIAS-SCAN)}$$

$$\frac{}{\Sigma \vdash \textbf{filter}\ a\ \overline{x}^{(n)} \Rightarrow \langle \overline{\emptyset}^{(n)} \rangle} \text{(ALIAS-FILTER)}$$

Figure 33: More aliasing rules.

single inference rule, which states that two occurrence traces can be sequentialized if and only if no array consumed in the left-hand trace is used in the right-hand trace.

Some of the most important inference rules for checking if an expression e is functionally safe with respect to in-place updates are shown in Figure 34, where the central judgment is $e \rhd \langle C, O \rangle$. We assume that the program is in strict A-normal form.

The rule for in-place update v_a **with** $[\overline{v}^{(n)}] \leftarrow v_v$ gives rise to an occurrence trace indicating that we have *observed* v_v and *consumed* v_a. Indices $\overline{v}^{(n)}$ are ignored as they are necessarily scalar variables, which cannot be consumed.

Another interesting rule concerns checking the safety of **map** expressions. We do not wish to permit the function of a **map** to consume any array bound outside of it, as that would imply the array may be consumed by more than one iteration of the **map**. However, the function may consume its *parameters*, which should be seen as the **map** expression as a whole consuming the corresponding input array. This restriction also preserves the parallel semantics of **map**, because different rows of a matrix can be safely updated in parallel. An example can be seen on Figure 36, which shows an in-place update nested inside a **map**. To express this restriction, we define an auxiliary judgment:

$$\mathcal{P} \vdash \langle C_1, O_1 \rangle \triangle \langle C_2, O_2 \rangle$$

Here, $\mathcal{P}$ is a mapping from parameter names to alias sets. Any variable v in O_1 that has a mapping in $\mathcal{P}$ is replaced with $\mathcal{P}[v]$ to produce O_2. If no such mapping exists, v is simply included in O_2. Similarly, any variable v in C_1 that has a mapping in $\mathcal{P}$ is replaced with the variables in the set $\mathcal{P}[v]$ (taking the union of all such replacements), producing C_2. However, if v does not have such a mapping, the judgment is not derivable. The precise inference rules are shown on Figure 35. Do-loops and function declarations can be checked for safety in a similar way. A function is safe with respect to in-place updates if its body consumes only those of the function's parameters that are unique (rule SAFE-FUN on Figure 35). For a function call, care is taken to ensure that the argument passed for a consumed parameter does not alias any other argument.

5.4 Related Work on Uniqueness Types

Futhark's uniqueness type system is similar to, but simpler, than the system found in Clean [BS93; BS96], where the primary motivation is modeling IO. Our use is more reminiscent of the ownership types of Rust [Hoa13]. The fact that Futhark is restricted in its support for user-defined data types, in particular not supporting pointer structures and instead emphasising the use of arrays, makes alias analysis tractable, and even possible to expose as a user-language feature.

While more powerful uniqueness type systems [BS93], and affine and linear types [TP11; FD02] are known, ours is the first application that directly addresses

Validity of sequencing $\boxed{\langle C_1, O_1\rangle \gg \langle C_2, O_2\rangle : \langle C_3, O_3\rangle}$

$$\frac{(O_2 \cup C_2) \cap C_1 = \emptyset}{\langle C_1, O_1\rangle \gg \langle C_2, O_2\rangle : \langle C_1 \cup C_2, O_1 \cup O_2\rangle} \quad \text{(OCCURENCE-SEQ)}$$

Uniqueness safety for expressions $\boxed{e \rhd \langle C, O\rangle}$

$$\frac{}{v \rhd \langle \emptyset, \text{aliases}(v)\rangle} \quad \text{(SAFE-VAR)}$$

$$\frac{}{k \rhd \langle \emptyset, \emptyset\rangle} \quad \text{(SAFE-CONST)}$$

$$\frac{e_1 \rhd \langle C_1, O_1\rangle \qquad e_2 \rhd \langle C_2, O_2\rangle \qquad \langle C_1, O_1\rangle \gg \langle C_2, O_2\rangle : \langle C_3, O_3\rangle}{\texttt{let}\ v_1\ \ldots\ v_n = e_1\ \texttt{in}\ e_2 \rhd \langle C_3, O_3\rangle\rangle} \quad \text{(SAFE-LETPAT)}$$

$$\frac{\begin{array}{c} v_1 \rhd \langle C_1, O_1\rangle \qquad e_2 \rhd \langle C_2, O_2\rangle \qquad e_3 \rhd \langle C_3, O_3\rangle \\ \langle C_1, O_1\rangle \gg \langle C_2, O_2\rangle : \langle C_2', O_2'\rangle \\ \langle C_1, O_1\rangle \gg \langle C_3, O_3\rangle : \langle C_3', O_3'\rangle \end{array}}{\texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rhd \langle C_2' \cup C_3', O_2' \cup O_3'\rangle} \quad \text{(SAFE-IF)}$$

$$\frac{}{v_a\ \texttt{with}\ [\overline{v}^{(n)}]\ \leftarrow v_v \rhd \langle \text{aliases}(v_a), \text{aliases}(v_v)\rangle} \quad \text{(SAFE-UPDATE)}$$

$$\frac{e_b \rhd \langle C, O\rangle \qquad \overline{p_i \mapsto \text{aliases}(v_i)}^{(n)} \vdash \langle C, O\rangle \triangle \langle C', O'\rangle}{\texttt{map}\ (\lambda \overline{p}^{(n)} : \overline{t}^{(m)} \to e_b)\ \overline{v}^{(n)} \rhd \langle C', O'\rangle} \quad \text{(SAFE-MAP)}$$

$$\frac{\begin{array}{c} \text{lookup}_{\text{fun}}(f) = \hat{\tau}_1 \to \cdots \to \hat{\tau}_n \to \hat{\tau}_r \\ \forall i.(\hat{\tau}_i = \star\tau) \Rightarrow (\forall j.j = i \vee v \notin \text{aliases}(v_j)) \\ C' = \bigcup_i \{\text{aliases}(v_i) \mid v_i \in \overline{v}^{(n)}, x_i = v \wedge \tau_i = \star\tau\} \\ O' = \bigcup_i \{\text{aliases}(v_i) \mid v_i \in \overline{v}^{(n)}\} - C' \end{array}}{f\ x_1\ \ldots\ x_n \rhd \langle C', O'\rangle} \quad \text{(SAFE-FUN)}$$

$$\frac{\begin{array}{c} e \rhd \langle C, O\rangle \\ \forall v \in C.\exists i.x_i = v \wedge \hat{\tau}_i = \star\tau \\ \forall i.(\hat{\tau}_i = \star\tau) \Rightarrow (\forall j.j = i \vee v \notin \text{aliases}(v_j)) \\ C' = \{\text{aliases}(v_i) \mid v_i \in \overline{v}^{(n)}, x_i = v \wedge \tau_i = \star\tau\} \\ O' = \{\text{aliases}(v_i) \mid v_i \in \overline{v}^{(n)}\} \setminus C' \end{array}}{\texttt{loop}\ (\overline{x_i : \hat{\tau}_i}^{(n)})\ =\ \overline{v}^{(n)}\ \texttt{for}\ z_1 < z_2\ \texttt{do}\ e \rhd \langle C', O'\rangle} \quad \text{(SAFE-LOOP)}$$

Figure 34: Checking safety of consumption.

Uniqueness safety for function definitions $\boxed{\triangledown fun}$

$$\frac{e \rhd \langle C, O\rangle \qquad \forall v \in C.\exists i.x_i = v \wedge \tau_i = \star\tau}{\triangledown \textbf{let}\ f\ \overline{x_i : \hat{\tau}_i}^{(n)}\ :\ \hat{\rho}_2\ =\ e} \quad \text{(SAFE-FUN)}$$

Validity of parameter consumption $\boxed{\mathcal{P} \vdash \langle C_1, O_1\rangle \triangle \langle C_2, O_2\rangle}$

$$\frac{}{\mathcal{P} \vdash \langle \emptyset, \emptyset\rangle \triangle \langle \emptyset, \emptyset\rangle} \quad \text{(OBSERVE-BASECASE)}$$

$$\frac{v \in \mathcal{P} \qquad \mathcal{P} \vdash \langle \emptyset, O\rangle \triangle \langle \emptyset, O'\rangle}{\mathcal{P} \vdash \langle \emptyset, \{v\} \cup O\rangle \triangle \langle \emptyset, \mathcal{P}[v] \cup O'\rangle} \quad \text{(OBSERVE-PARAM)}$$

$$\frac{v \notin \mathcal{P} \qquad \mathcal{P} \vdash \langle \emptyset, O\rangle \triangle \langle \emptyset, O'\rangle}{\mathcal{P} \vdash \langle \emptyset, \{v\} \cup O\rangle \triangle \langle \emptyset, \{v\} \cup O'\rangle} \quad \text{(OBSERVE-NONPARAM)}$$

$$\frac{v \in \mathcal{P} \qquad \mathcal{P} \vdash \langle C, O\rangle \triangle \langle C', O'\rangle}{\mathcal{P} \vdash \langle \{v\} \cup C, O\rangle \triangle \langle \mathcal{P}[v] \cup C', O'\rangle} \quad \text{(CONSUME-PARAM)}$$

Figure 35: Checking parameter consumption.

```
-- This one is OK and considered to consume 'as'.
let bs = map (λa → a with [0] ← 2) as
let d  = iota m
-- This one is NOT safe, since d is not a formal parameter.
let cs = map (λi → d with [i] ← 2) (iota n)
```

Figure 36: Examples of **map**s with in-place updates.

`map`-style parallel constructs, and shows how in-place updates can be supported without making evaluation order observable. Our system is also somewhat more "friendly", as the former approaches deal with more complicated systems and rely on significantly more complex analysis, where for example aliasing of affine variables is banned, while in our case unique arrays may be used multiple times before being consumed, with aliasing information used to flag the use-after-consumption cases. Indeed, the uniqueness type system shown here only affects function definitions and function calls, with exclusively aliasing analysis used for the intraprocedural case.

Some prior work does exist for destructively updating memory in the presence of bulk parallel operators. For example, [Les09] discusses fusing away intermediate copies when composing bulk updates in Haskell. In contrast, our uniqueness types are (i) intended for sequential parts of the code, and (ii) able to guarantee the absense of any copying whatsoever. For example in the above cited work, the operation `xs//vs1/vs2`, which first updates `xs` with `vs1` and then updates the result with `vs2`, is fused to perform just a single copy of `xs` instead of two, but the copy of `xs` cannot be avoided, because there is no way in Haskell to enforce that some array variable may no longer be used.

Chapter 6

Size Inference

A great many optimisations and safety checks in Futhark depend on how the shape of two arrays relate to each other, or at which point the shape of an array can be known (especially if that point is much earlier than the point at which the *values* of the array can be known). Especially the latter is important for the moderate flattening transformation (Chapter 8) that constitutes one of the main contributions of the thesis. Nested parallelism supports the construction of arrays whose values are dependent on some outer parallel construct. However, for *regular* nested parallelism, the shapes of those arrays can be computed invariant to all parallel loops. It is crucial for efficient execution that we can hoist the computation of such sizes out of parallel loops. This requires us to reify the notion of an array shape, and the computation of that shape, in the IR. In the Futhark compiler, we treat size computations like any other expression, which allows us to use our general compiler optimisation repertoire to optimise and simplify the computation of sizes.

The main contribution of this chapter is an IR design that maintains the invariant that for *any* array-typed variable in scope, each dimension of that array corresponds to some `i32`-typed variable also in scope. For expressions where the shape of the result cannot be computed in advance (consider a **`filter`** or a function call), we use a lightweight mechanism based on *existential types*. This chapter also discusses how we move from the unsized IR to the sized IR (analogous to "typed" versus "untyped"), and how most size calculations can subsequently be optimised away. An important technique is *function slicing*, which we use to derive functions that precompute the size of values returned by a function.

Section 6.1 introduces the syntax of the IR, which is a slight extension of the one presented in the previous chapter. In Section 6.2 we give an intuition for how size inference is accomplished, via an example of how a program progresses from having no inferred sizes, to being highly existential, to finally having most sizes resolved statically. Section 6.3 presents a subset of the type rules for the sized IR, and Section 6.4 shows some of the transformations used to add size information to an unsized program.

$$
\begin{array}{lll}
\tau & ::= & \mathtt{t} \mid [d]\,\tau \qquad \text{(Scalar/array type)} \\
\phi & ::= & (x_1 : \tau_1) \to \cdots \to (x_n : \tau_n) \to \exists \bar{d}.\rho \qquad \text{(Sizes of results} \in \bar{d}) \\
\underline{\tau} & ::= & \alpha \mid (x_1 : \tau_1, \ldots, x_n : \tau_n) \mid (x : \underline{\tau}) \to \exists \bar{d}.\underline{\tau} \qquad \text{(Sizes of results} \in \bar{d}) \\
\underline{\phi} & ::= & \forall \bar{\alpha}.\phi \\
\text{fun} & ::= & \mathtt{let}\ f\ \hat{p}\ :\ \overline{d}.\hat{\tau}\ =\ e \qquad \text{(Sizes of results} \in \bar{d})
\end{array}
$$

Figure 37: Types with embedded size information and named parameters. The remaining syntax definitions remain unchanged, except that they refer to the new definitions above.

6.1 A Sized IR

One important observation of the IR presented in the previous chapter is that some operator-semantics invariants, related to the array regularity, are guaranteed to hold by construction, but several other invariants are only "assumed", that is, they have not been verified (made explicit) in the IR:

- **`iota`** and **`replicate`** assume a non-negative first argument, and the size of the resulting array is exactly the value of the first argument.
- **`map`** is guaranteed to receive arguments of identical outermost size, which also matches the outermost size of all result arrays[1]. However, **`map`** assumes that its function argument produces arrays of identical shape for each element of the input array.
- **`filter`** receives and produces arguments and results of identical outermost size, respectively (and the outermost size of the argument is not smaller than the one of the result).
- **`reduce`** and **`scan`** receive arguments of identical outermost size, and **`scan`** results have outermost size equal to that of the input. The semantics for **`reduce`** and **`scan`** assumes that the two arguments and result of the binary associative operator have identical shapes.

Figure 37 shows an extended type system in which (i) sizes are encoded in each array type, that is, $[d]\tau$ represents the type of an array in which the outermost dimension has size d, and in which (ii) function/lambda types use universal quantifiers for the sizes of the array parameters ($\forall s_1$), and existential quantifiers for the sizes of the result arrays ($\exists s_2$). Function types now also contain *named* parameters, supporting a simple variant of dependent types. For function parameters where the name is irrelevant, we shall elide the name and use the same notation as before. Figure 38

[1] In the user language **`zip`** accepts an arbitrary number of array arguments that are required to have the same outermost size.

op		TySch(*op*)
iota	:	$(d : \texttt{i32}) \to [d]\texttt{i32}$
replicate	:	$\forall\alpha.(d : \texttt{i32}) \to \alpha \to [d]\alpha$
reshape	:	$\forall\overline{d}^{(m)}\alpha.(x_1 : \texttt{i32}, \ldots, x_n : \texttt{i32}) \to [d_1]\cdots[d_m]\alpha \to [x_1]\cdots[x_n]\alpha$
rearrange $(\overline{c}^{(n)})$	:	$\forall\overline{d}^{(n)}\alpha.[d_1]\cdots[d_n]\alpha \to [d_{p(1)}]\cdots[d_{p(n)}]\alpha$ where $p(i)$ is the result of applying the permutation induced by $c_1, \ldots, c_n$.
map	:	$\forall d\ \bar{\alpha}^{(n)}\bar{\beta}^{(m)}.$ $(\overline{\alpha}^{(n)} \to (\bar{\beta}^{(m)})) \to \overline{[s]\alpha_i}^{(n)} \to (\overline{[s]\beta_i}^{(m)})$
scatter	:	$\forall d\overline{x}^{(m)}\overline{\alpha}^{(n)}.$ $(\overline{[x_i]\alpha_i}^{(n)}) \to (\overline{[d]\texttt{i32}}^{(n)}) \to (\overline{[d]\alpha_i}^{(n)}) \to (\overline{[x_i]\beta_i}^{(m)})$
reduce	:	$\forall d\ \bar{\alpha}^{(n)}.$ $(\overline{\alpha}^{(n)} \to \overline{\alpha}^{(n)} \to (\bar{\beta}^{(n)})) \to (\overline{\alpha}^{(n)}) \to \overline{[d]\alpha_i}^{(n)} \to (\overline{\alpha}^{(n)})$
scan	:	$\forall d\ \bar{\alpha}^{(n)}.$ $(\overline{\alpha}^{(n)} \to \overline{\alpha}^{(n)} \to (\bar{\beta}^{(n)})) \to (\overline{\alpha}^{(n)}) \to \overline{[d]\alpha_i}^{(n)} \to (\overline{[d]\alpha_i}^{(n)})$
filter	:	$\forall d_1\ \bar{\alpha}^{(n)}.$ $(\overline{\alpha}^{(n)} \to \texttt{bool}) \to \overline{[d_1]\alpha_i}^{(n)} \to \exists d_2.(\overline{[d_2]\alpha_i}^{(n)})$
stream_map	:	$\forall d\ \overline{\alpha}^{(n)}\ \overline{\beta}^{(m)}.$ $((x : \texttt{i32}) \to \overline{[x]\alpha_i}^{(n)} \to (\overline{[x]\beta_i}^{(m)})) \to \overline{[d]\alpha_i}^{(n)} \to (\overline{[d]\beta}^{(m)})$
stream_red	:	$\forall d\ \overline{\alpha}^{(n)}\overline{\beta}^{(m)}.(\overline{\beta}^{(m)} \to \overline{\beta}^{(m)} \to (\overline{\beta}^{(m)})) \to ((x : \texttt{i32}) \to \overline{[x]\alpha_i}^{(n)} \to (\overline{\beta_i}^{(m)})) \to \overline{[d]\alpha_i}^{(n)} \to (\overline{\beta}^{(m)})$

Figure 38: Dependent-size types for various SOACs.

also shows that this extension allows to encode most of the afore-mentioned invariants into size-dependent types. The requirement for non-negative input to **iota** and **replicate** remains only dynamically checked. We see that most parameters remain unnamed, but are crucially used to encode the shape properties of **iota** and **replicate**.

The type of **map** is interesting because the result array types do not follow immediately from the input array types. Instead, it is expected that the functional argument declares the result type (including sizes) in advance. Operationally, we can see this as being able to "pre-allocate" space for the result. However, the return size cannot in general be known in advance without evaluating the function. We shall return to this issue in Section 6.4.4.

The main difference between the size-dependent typing of the core language, and the size annotations of the source language, is that the latter are optional and checked dynamically, while the former are mandatory and enforced statically. As shown on

```
let concat (xs: []f64) (ys: []f64): []f64 =
  let a = size 0 xs
  let b = size 0 ys
  let c = a+b
  let (is: []i32) = iota c
  let (os: []f64) =
    map (λi → if i < a then xs[i] else ys[i+b]) is
  in os

let f (vss: [][]f64): []f64 =
  let (ys: []f64) (zs: []f64) =
    map (λ(vs: []f64) →
          let ys = reduce (+) 0.0 vs
          let zs = reduce (*) 1.0 vs
          in (ys, zs))
        vss
  let (rs: []f64) = concat ys zs
  in rs

let main (vsss: [][][]f64): [][]f64 =
  let (rss: [][]f64) =
    map (λ(vss: [][]f64): []f64 →
           let rs = f vss
           in rs) vsss
  in rss
```

Figure 39: Running example: Program in un-sized IR.

Figure 38, the **reshape** construct functions as an "escape hatch", by which we can arbitrarily transform the shape of an array. An implicit run-time check enforces that the original shape and the desired shape has the same total number of elements. This is used to handle cases that would otherwise not be well-typed.

6.2 Size Inference by Example

This section demonstrates, by example, the code transformation that (i) makes explicit in the code the shape-dependent types and verifies the assumed invariants and (ii) optimizes away in many cases the existential types. Our philosophy is to initially use existential types liberally, with the expectation that inlining and simplification will eventually remove almost all of them.

The program in Figure 39 receives as input a three-dimensional array `vsss`, and

```
let concat (n: i32) (m: i32) (xs: [n]f64) (ys: [m]f64)
           : d.[d]f64 =
  let a = n
  let b = m
  let c = a+b
  let (is: [c]i32) = iota c
  let (os: [c]f64) =
    map (λi → if i < n then xs[i] else ys[i+a]) is
  in os

let f (m: i32) (k: i32) (vss: [m][k]f64): d.[d]f64 =
  let (ys: [m]f64) (zs: [m]f64) =
    map (λ(vs: [k]f64) →
          let ys = reduce (+) 0.0 vs
          let zs = reduce (*) 1.0 vs
          in (ys, zs))
        vss
  let (l: i32) (rss: [l]f64) = concat m m ys zs
  in rs

let main (n: i32) (m: i32) (k: i32)
         (vsss: [n][m][k]f64): d1 d2.[d1][d2]f64 =
  let l = if n != 0
          then let (d: i32) (ws: [d]f64) = f n m vsss[0]
               in d
          else 0
  let (rss: [n][d]f64) =
    map (λ(vss: [m][k]f64): [l]f64 →
          let (d: i32) (w: [d]f64) = f m k vss
          let vs = reshape l w
          in vs)
        vsss
  in rss
```

Figure 40: Running example: ∃-quantified target IR. Changes compared to Figure 39 highlighted in red.

produces a two-dimensional array, by mapping the elements of the outermost dimension of `vsss` by function `f`.

The first stage, demonstrated in Figure 40, transforms the program into an unoptimised version in which (i) all arrays have shape-dependent types, which may be existentially quantified, and (ii) all "assumed" invariants are explicitly checked. This

is achieved by:

- Extending the function signatures to encompass also the shape information for each array argument. For example, `f` takes additional parameters `m` and `k` that specify the shape of array argument `vss`,
- Representing function's array results via existentially-quantified shape-dependent types. For example, the return type of `f` is specified as `d.[d]f64`, indicating an existential size `d`.
- Modifying **`let`** patterns to also explicitly bind any existential sizes returned by the corresponding expression. For example, the binding of the result of `concat` now also includes a variable `l`, representing the result size.
- For the **`map`** in `main`, we need to make a "guess" at the size of the array being returned, which we store as the variable `l`. This guess is made by applying the **`map`** function to `vsss[0]`, which produces both a size and an array, from which we use just the size. This corresponds to *slicing* the **`map`** function. If `vsss` is empty (that is, if `n` is zero), the guess is zero. There is a risk here: if the **`map`** function consumes any of its parameters via in-place updates, the slice will likewise, and since `vsss[0]` aliases `vsss`, we would end up consuming `vsss` twice. As a result, we must copy any inputs that are consumed in the sliced function, although for this example, there are none.

  Since `f` returns an existential result, and the lambda *must* return an array of exactly type `[l]f64`, we use a **`reshape`** to obtain this desired type. Since the **`reshape`** fails if `d != n`, this effectively ensures that the **`map`** produces a regular array.
- Since all arrays in scope also have variables in scope for describing their size, replace all uses of **`size`** with references to those variables.

It is important to note that this transformation preserves asymptotically the number of operations of the original program. However, it performs a significant amount of redundant computation. To compute `l`, we compute the entire result, only to throw most of it away. General-purpose optimisation techniques can be employed to eliminate the overhead. On Figure 41 we see the result of inlining all functions, followed by straightforward simplification, dead code removal, and hoisting of the computation of `c`. The result is that all arrays constructed inside the **`map`**s have a size that can be computed before the **`map`**s are entered. From an operational perspective, this lets us pre-allocate memory before executing the **`map`**s on a GPU. This is essential for GPU execution because dynamic allocation and assertions are typically not well suited for accelerators, hence the shapes of the result and of various intermediate arrays need to be computed (or at least overestimated) and verified before the kernel is run.

```
let main (n: i32) (m: i32) (k: i32)
         (vsss: [n][m][k]f64): d1 d2.[d1][d2]f64 =
  let c = m+m
  let l = if n != 0 then c else 0
  let (rss: [n][d]f64) =
    map (λ(vss: [m][k]f64): [l]f64 →
          let (ys: [m]f64) (zs: [m]f64) =
            map (λ(vs: [k]f64) →
                  let ys = reduce (+) 0.0 vs
                  let zs = reduce (*) 1.0 vs
                  in (ys, zs))
                vss
          let (is: [c]i32) = iota c
          let (os: [c]f64) =
            map (λi → if i < n then xs[i] else ys[i+a])
                is
          let rs = reshape l os
          in rs)
        vsss
  in rss
```

Figure 41: The running example after inlining all functions and performing simple inlining, CSE, dead-code elimination, and simplification. No existential quantification is left, except in the return type of the main function.

However, there is still a problem with the code shown on Figure 41. The issue is that the compiler cannot statically see that `l==c`, and thus has to maintain the **`reshape`** and perform a dynamic safety check at run-time. This is because the computation of `l` is hidden behind a branch. The branch was conservatively inserted because we could not be sure that the value of `vsss[0]` (on Figure 40) would not be used for computing the size (size analysis is intraprocedural, and so by the time we first computed the slize, we had no insight in the definition of `f`). The branch was inserted for safety reasons, but is now a hindrance to further simplification. There are at least two possible solutions, both of which are used by the present Futhark compiler:

1. Give the programmer the ability to annotate the original lambda (in the source language) with the return type, *including* expected size. This effectively lets the programmer make the guess for us, with no branch required. The result is still checked by a **`reshape`**, but in most cases the guess will be statically the same as the computed size, and the **`reshape`** can thus be simplified away.

2. Somehow "mark" the branch as being a size computation. Then, after inlining and simplification, we can recognise such branches, and simplify them to their "true" branch, if that branch contains only "safe" expressions, where a safe expression is one whose evaluation can never fail. We have to wait until after inlining and simplification, as a function call can never be considered safe.

   This solution has the downside that it may affect whether an inner size of an empty array is zero or nonzero.

In practice, the first solution is preferable in the vast majority of cases, as it also serves as useful documentation of programmer intent in the source program.

A third solution is to factor out the "checking" part of the **`reshape`** operation. This approach is sketched on Figure 42. Here, we use a hypothetical `assert` expression for computing a "certificate", on which the **`reshape`** expression itself is predicated. To enable the check to be hoisted safely out of the outer **`map`**, the condition also succeeds if the outer map contains no iterations (`n == 0`). The Futhark compiler currently makes only limited use of this technique, as the static equivalence of sizes is a more powerful enabler of other optimisations.

One may observe that in the resulting code, the shape and regularity of `rss` are computed and verified before the definition of `rss`, respectively, and, most importantly, that the size computation and assumed-invariant verification in this case introduces negligible overhead, i.e., $O(1)$ number of operations.

Note that the code shown on Figure 40 is the only *required* step we have to perform. Subsequent optimisation to eliminate existential quantification could be done in any way that is found desirable, perhaps via interprocedural analysis or slicing. Previously, we experimented with a technique based on *slicing*, where a function `g` is divided into two functions `g_shape` and `g_value`, the first of which computes the sizes of all (top-level) arrays in the latter, including the result [HEO14]. An example is shown on Figure 43, which contains a portion of the running example. The `concat` function has been split into `concat_shape` and `concat_value`. The call to `concat` has been likewise split.

In practice, sophisticated dead code removal may be necessary to obtain efficient shape functions — for example, we will need to remove shape-invariant loops — and our approach thus requires the compiler to possess an effective simplification engine.

The Futhark compiler currently does not use this approach of splitting functions. This is partly because of the risk of asymptotic slowdown in the presence of recursion (which was supported at the time), and partly because merely inlining plus simplification is easier to implement, and performed equally well for our purposes.

6.3 New Type Rules

The addition of size-dependent types requires an extension of the typing rules. The main problem is the handling of an existential context in the type of an expression.

```
let main (n: i32) (m: i32) (k: i32)
         (vsss: [n][m][k]f64): d1 d2.[d1][d2]f64 =
  let l = if n != 0 then m+m else 0
  let c = m+m
  let cert = assert(n == 0 || l==c)
  let (rss: [n][d]f64) =
    map (λ(vss: [m][k]f64): [l]f64 →
          let (ys: [m]f64) (zs: [m]f64) =
            map (λ(vs: [k]f64) →
                  let ys = reduce (+) 0.0 vs
                  let zs = reduce (*) 1.0 vs
                  in (ys, zs))
                vss
          let (is: [c]i32) = iota c
          let (zs: [c]f64) =
            map (λi → if i < n then xs[i] else ys[i+a]) is
          let rs = reshape<cert> l zs
          in rs)
        vsss
  in rss
```

Figure 42: Separating the size-checking of a **reshape** from the **reshape** itself.

First, we define a subtyping relation on existential types. The judgment and its associated rules is shown on Figure 44. Apart from the usual rules for reflexivity and transitivity, we define alpha substitution (it is valid to change the names bound by the existential context), and *weakening*. In our context, weakening corresponds to making the size of an array type less concrete. For example, we can weaken the type $[x]\texttt{i32}$ to $\exists x.[x]\texttt{i32}$ (we would usually also rename x to avoid confusion). The need for weakening arises in particular due to the typing rules for **let** and **if**, as we shall see.

We extend the type judgment for expressions so that it now returns a type with an existentially quantified part (which may be empty). The most interesting rules are shown on Figure 45 and discussed below. We ignore uniqueness attributes here, as they have no influence on size-dependent typing. The rule for array operators performs some abuse of notation to construct the substitution S; the intent is to construct a substitution from names used in the type scheme (even the $\forall$-quantified context) to the operator arguments supplying the concrete value. This is used in for example the size-dependent typing of **iota**.

The first rule states that if we can derive expression e to have some existential type $\exists\overline{d}.\rho$, then e also has any existential type that is a supertype of what we derived.

```
let concat_shape (n: i32) (m: i32)
                 (xs: [n]f64) (ys: [m]f64): i32 =
  n+m

let concat_value (n: i32) (m: i32) (c: i32)
                 (xs: [n]f64) (ys: [m]f64): [c]f64 =
  let (is: [c]i32) = iota c
  let (zs: [c]f64) =
    map (λi → if i < n then xs[i] else ys[i+n]) is
  in zs

let f (m: i32) (k: i32) (vss: [m][k]f64): d.[d]f64 =
  let (ys: [m]f64) (zs: [m]f64) =
    map (λ(vs: [k]f64) →
          let ys = reduce (+) 0.0 vs
          let zs = reduce (*) 1.0 vs
          in (ys, zs))
        vss
  let (l: i32) = concat_shape m m c ys zs
  let (rs: [l]f64) = concat_value m m c ys zs
  in rs
```

Figure 43: An example of applying the slicing approach to `concat`.

This rule is what allows us to weaken the typing for an expression, which is necessary in some cases cases, as we shall see below.

The rule for **`let`** requires that the names bound may not be present in the type of the returned value. As an example, consider an expression **`replicate`** `x v`. Supposing $\texttt{v} : \tau$, this expression has type $[x]\tau$ while **`let`** `x = y` **`in replicate`** `x v` has type $\exists d.[d]\tau$ (with d picked arbitrarily). It is crucial that we are able to use weakening to loosen the type of the body of the **`let`**-binding, or else we could not type it.

It is also in the rule for **`let`**-bindings that we bind the existential sizes to concrete variables. For example, if the return type of the function contains l existential sizes, then we require that the pattern begins with l names of type `i32`. Operationally, these will be bound to the actual sizes of the value returned by the function. As a slight hack, we require that these names match exactly the corresponding existential context. We can always use alpha substitution to ensure a match. For example, given a function

$$\texttt{f} : (\texttt{x} : \texttt{i32}) \to (\texttt{y} : \tau) \to \exists \texttt{n}.[\texttt{x}][\texttt{n}]\tau$$

Existential subtyping $\boxed{\exists\overline{x}.\rho' <: \exists\overline{y}.\rho}$

$$\frac{}{\exists\overline{x}^{(n)}.\rho <: \exists\overline{x}^{(n)}.\rho} \tag{6.1}$$

$$\frac{\exists\overline{z}^{(l)}.\rho'' <: \exists\overline{y}^{(m)}.\rho' \quad \exists\overline{y}^{(m)}.\rho' <: \exists\overline{x}^{(n)}.\rho}{\exists\overline{z}^{(l)}.\rho'' <: \exists\overline{x}^{(n)}.\rho} \tag{6.2}$$

$$\frac{\begin{array}{c}S \text{ is bijective from } \overline{x}^{(n)} \text{ to } \overline{y}^{(n)} \\ \text{None of } y \text{ used free in } \rho.\end{array}}{\exists\overline{y}^{(n)}.S(\rho) <: \exists\overline{x}^{(n)}.\rho} \tag{6.3}$$

$$\frac{y \text{ used free in } \rho}{\exists\ y\ \overline{x}^{(n)}.\rho <: \exists\ \overline{x}^{(n)}.\rho} \tag{6.4}$$

Figure 44: The subtyping relationship for existential types. The four rules describe reflexivity, transitivity, renaming, and weakening.

we can derive a typing judgment for the **let**-binding

$$\textbf{let}\ (\texttt{m} : \texttt{i32}, \texttt{v} : [\texttt{a}][\texttt{m}]\tau) = \texttt{f a b}\ \textbf{in}\ e$$

if we assume that `a : i32` and e is well-typed, but

$$\textbf{let}\ (\texttt{m} : \texttt{i32}, \texttt{v} : [\texttt{m}][\texttt{a}]\tau) = \texttt{f a b}\ \textbf{in}\ e$$

is a type error (note the swapped dimensions in the type of `v`). However, we can always further weaken the type and make all sizes existential, as in

$$\textbf{let}\ (\texttt{m} : \texttt{i32}, \texttt{k} : \texttt{i32}, \texttt{v} : [\texttt{m}][\texttt{k}]\tau) = \texttt{f a b}\ \textbf{in}\ e$$

and in fact this is how size information is initially inserted by the size inference procedure described in Section 6.4.

Using weakening, a pattern can also contain existential parts that are not immediately existential in the type. This is useful for supporting gradual simplification of existential sizes, without having the intermediate steps be ill-typed For example, consider this expression:

```
let (d: i32) (a: [d]i32) = (let y = x in replicate y 0) in e
```

Expressions $\boxed{\Gamma \vdash e : \exists \overline{d}.\rho}$

$$\frac{\Gamma \vdash e : \exists \overline{x}.\rho \quad \exists \overline{y}.\rho' <: \exists \overline{x}.\rho}{\Gamma \vdash e : \exists \overline{y}.\rho'} \tag{6.5}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{p}} p : (\overline{d_i : \mathtt{i32}}^{(l)}, \rho) \quad \Gamma \vdash e_1 : \exists \overline{d}^{(l)}.\rho \\ \Gamma, p \vdash e_2 : \exists \overline{d^r}^{(k)}.\rho' \\ \text{No name bound by } p \text{ used in } \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ p = e_1\ \mathtt{in}\ e_2 : \exists \overline{d^r}^{(k)}.\rho'} \tag{6.6}$$

$$\frac{\begin{array}{c}\mathrm{lookup}_{\mathrm{fun}}(f) = \overline{(p_i : \tau_i)}^{(n)} \to \exists \overline{d}^{(l)} \overline{\tau'}^{(m)} \\ S = \langle \overline{p_i \mapsto x_i}^{(n)} \rangle \quad \forall i \in \{1, \ldots, n\}.\Gamma(x_i) = S(\tau_i)\end{array}}{\Gamma \vdash f\ x_1\ \ldots\ x_n : \exists \overline{d}^{(l)}.S(\overline{\tau'}^{(m)})} \tag{6.7}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{a}} a_i : \underline{\tau}_i^p \quad i \in \{1, \ldots, n\} \\ \mathrm{TySch}(op) = \overline{(p_i : \underline{\tau}_i)}^{(n)} \to \exists \overline{d}^{(l)}.\overline{\tau'}^{(m)} \\ S = \langle \overline{p_i \mapsto x_i}^{(m)} \mid \text{where } x_i = a_i \rangle \quad \forall i \in \{1, \ldots, n\}.\underline{\tau}_i^p = S(\underline{\tau}_i)\end{array}}{\Gamma \vdash op\ \overline{a}^{(n)} : \exists \overline{d}^{(l)}.S(\overline{\tau'}^{(m)})} \tag{6.8}$$

$$\frac{\begin{array}{c}\Gamma(s) = \mathtt{bool} \\ \Gamma \vdash e_1 : \exists \overline{x}.\rho \quad \Gamma \vdash e_2 : \exists \overline{x}.\rho\end{array}}{\Gamma \vdash \mathtt{if}\ s\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \exists \overline{x}.\rho} \tag{6.9}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{p}} p : \exists \overline{d}^{(n)}.\rho \quad \Gamma \vdash e_1 : \exists \overline{d}^{(n)}.\rho \quad \Gamma \vdash e_3 : \exists \overline{d}^{(n)}.\rho \\ \Gamma \vdash e_2 : \mathtt{i32}\end{array}}{\Gamma \vdash \mathtt{loop}\ \overline{(d_i : \mathtt{i32})}^{(n)}\ p = e_1\ \mathtt{for}\ x < e_2\ \mathtt{do}\ e_3 : \exists \overline{d}^{(n)}.\rho} \tag{6.10}$$

Figure 45: Size-aware typing rules for particularly interesting expressions. The subtyping relationship is defined on Figure 44.

Since the size `y` is bound inside of the outer **`let`**-binding, we have no choice but to leave the size `d` existential. If the compiler then performs copy-propagation, we obtain the following expression:

`let` `(d: i32) (a: [d]i32) =` **`replicate`** `x 0` **`in`** *e*

The type of **`replicate`** `x 0` is `[x]i32`, but since copy-propagation is not concerned with modifying patterns in **`let`**-bindings, we still existentially quantify the size of the result. A subsequent simplification can then be applied to remove unnecessary existential parts of patterns, by noting that `d` will always be `x`, yielding:

`let` `(a: [x]i32) =` **`replicate`** `x 0` **`in`** *e*

The vast majority of the existential-related simplification we perform is of the trivial nature above, where gradual rewrites eventually bring the program to a form where the existential sizes can be statically resolved.

From a compiler engineering point of view, it would be awkward if copy-propagation (or other optimisations) were also responsible for fixing any changes to existential sizes that were caused by their simplifications. It is cleaner to separate this into multiple separate transformations, but this requires that the intermediate forms are well-typed. This fits well with the general philosophy in the Futhark compiler of gradual refinement, as was also shown in the previous section. This leniency will also help us when first inserting the existential information, as shown in the next section. However, making a type *more* existential would still require us to fix up the relevant patterns.

Another interesting case is for functions and operators, due to named parameters. We assume that all arguments to the function are variable names. Intuitively, we then construct a substitution S_p from the parameter names in the type of the function to the concrete names of the arguments, and apply this substitution before checking whether the argument types match the parameter types. The substitution is also applied to the result type.

We use a similar trick for **`if`** expressions, where both branches must return existential types with identical existential contexts. For example, consider the following expression:

`if` *c* **`then`** **`replicate`** *x* (**`iota`** *y*) **`else`** **`replicate`** *y* (**`iota`** *x*)

The "true" branch has type $[x][y]\texttt{i32}$, while the "false" branch has type $[y][x]\texttt{i32}$. The least existential type that can encompass both of these is $\exists nm.[n][m]\texttt{i32}$, which is then the type of the **`if`**-expression.

6.4 Size Inference

This section presents a set of syntax-directed rules for transforming an un-annotated Futhark program into an *annotated* Futhark program, where all types have size infor-

mation. Concretely, we are performing a syntax-directed translation that carries out the following tasks:

1. Add existential contexts to function return types to match the number of (distinct) array dimensions.
2. Add extra parameters to all functions corresponding to the number of (distinct) array dimensions in their parameters.
3. Add existential contexts to all **`let`**-patterns corresponding to the existential type of the bound expression.
4. Insert **`reshape`** expressions where necessary (such as for inputs to **`map`**) to ensure that size-related restrictions are maintained.
5. Amend any type $[]\tau$ such that it has form $[d]\tau$, where d is some variable in scope. After the preceding steps have been accomplished, this can be done by looking at the context in which the type occurs and following the type rules.

We will ignore the distinction between unique and non-unique types, as these have no influence on size inference. We write $\mathrm{d}(\tau)$ to indicate the *rank* (number of dimensions) of a type τ. We may also write $\mathrm{d}(v)$, where v is not itself a type, but something which *has* a type, such as a variable.

6.4.1 Fundamental Transformation

For each function f in the original program, we generate the *existential function* f_{ext}, that returns the values returned by f, with all shapes in the return type being existentially quantified.

Specifically, if the return type of a function f is $\overline{\tau}^{(n)}$, then the return type of f_{ext} is $\exists \overline{d_i^1}^{(\mathrm{d}(\tau_1))} \cdots \overline{d_i^n}^{(\mathrm{d}(\tau_n))}.(\tau'_1, \ldots, \tau'_n)$, where $\mathrm{d}(\tau)$ is the rank of τ, and τ'_j is τ_j shape-annotated with $\overline{d_i^j}^{(\mathrm{d}(\tau_j))}$. For example, if f returns type `[][]i32`, f_{ext} will return type $\exists d_1\ d_2[d_1][d_2]$`i32`. Thus, after transformation, the shape of the return of a function will be existentially quantified.

Furthermore, the parameters of f are likewise annotated. An explicit `i32` parameter is added for every dimension of an array parameter, with the array parameter itself annotated to refer to the corresponding `i32` parameter. For example, if f takes a single parameter `[][]i32`p, then f_{ext} will take three parameters `i32` n, `i32` m, and $[m][n]$`i32`

In this chapter we assume for simplicity that the original program contains no pre-existing size information, as by the un-sized IR presented in the previous chapter. This is not congruent with the actual Futhark source language, where user-provided

shape invariants may be present. In the implementation, these are handled by imposing constraints on the dimensions we generate. For example, if the source program contains information relating certain dimensions of the function return type to specific function parameters, we do not generate existential dimensions, but instead refer directly to the function parameters. Likewise, if the source program specifies that certain dimensions are identical to each other, we simply generate a single size parameter and use that for all the dimensions in question.

We will use the following function $\mathcal{T}(\tau\ x, s)$ to annotate a type τ with the shapes in s, which is a sequence of variable names whose length is equal to the rank of τ:

$$\mathcal{T}(\texttt{[]}_1 \cdots \texttt{[]}_n \texttt{t}, \overline{s}^{(n)}) = \texttt{[}s_1\texttt{]} \cdots \texttt{[}s_n\texttt{]t}$$

As an example:

$$\mathcal{T}(\texttt{[][]i32}, \texttt{n}, \texttt{m}) = \texttt{[m][n]i32}$$

6.4.2 Transformation Functions

The function $\mathcal{A}^{\text{exp}}_{\Sigma}(b)$ computes the *annotated* version of the body b in the environment Σ and returns shapes as well as values (that is, its type will contain existentials). The environment Σ is a mapping from array names to lists of variable names, where element i of the list denotes the size of dimension i of the corresponding array. We will use conventional head, tail and drop operations to manipulate these lists, as well as bracket notation for arbitrary indexing; we write the "cons" operation as $x :: xs$.

The function $\mathcal{A}^{\text{fun}}(f)$ computes the existentially-quantified function f_{ext}, by using $\mathcal{A}^{\text{exp}}_{\Sigma}$ to annotate the function's body (and result) with shapes information, and by modifying the function's type as described in the beginning of Section 6.4.1.

We also define a function $\mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \overline{r}, \overline{p})$. This is similar to $\mathcal{A}^{\text{fun}}(f)$, except that (i) we work within an environment Σ, and (ii) we know in advance the intended shape of the result ($\overline{r}$) and parameters ($\overline{p}$), because the result shape of an anonymous function is never existentially quantified.

6.4.3 Simple Rules

This section describes cases for the function $\mathcal{A}^{\text{exp}}_{\Sigma}(e)$. The core rules are listed in Figure 46, including the rule for **`let`**-patterns, which is likely the most important one. Here we create an existential size for *every* dimension of the bound variables. It is assumed that subsequent simplification will remove those that can be resolved statically.

Observe how `size` expressions are completely removed from the annotated program, and instead replaced with the variable storing the desired size. The rule for **`if`**

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let}\ \overline{v_i : \tau_i}^{(n)} = e_1\ \texttt{in}\ e_2) =$$
$$\quad \texttt{let}\ \overline{\text{sizes}(\tau_i)}^{(n)}\ \overline{v_i : \mathcal{T}(\tau_i, \text{sizes}(\tau_i))}^{(n)} = \mathcal{A}^{\text{exp}}_{\Sigma}(e_1)\ \texttt{in}\ \mathcal{A}^{\text{exp}}_{\Sigma'}(e_2)$$
$$\text{where}\ \ \text{sizes}(\tau_j) = \overline{d^j_i}^{(\text{d}(\tau_j))}$$
$$\Sigma' = \overline{v_i \mapsto \text{sizes}(\tau_i)}^{(n)}, \Sigma$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{let}\ (v : \texttt{i32}) = \texttt{size}\ k\ a\ \texttt{in}\ e) =$$
$$\quad \texttt{let}\ (v : \texttt{i32}) = \Sigma(a)[k]\ \texttt{in}\ \mathcal{A}^{\text{exp}}_{\Sigma}(e)$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) =$$
$$\quad \texttt{if}\ \mathcal{A}^{\text{exp}}_{\Sigma}(e_1)\ \texttt{then}\ \mathcal{A}^{\text{exp}}_{\Sigma}(e_2)\ \texttt{else}\ \mathcal{A}^{\text{exp}}_{\Sigma}(e_3)$$

$$\mathcal{A}^{\text{exp}}_{\Sigma}(f\ \overline{x}^{(n)}) =$$
$$\quad f_{ext}\ \overline{\Sigma(x_i)}^{(n)}\ \overline{x}^{(n)}$$

Figure 46: Simple transformation rules for expressions.

is completely straightforward, and simply applies the transformation to the subexpressions. Most of the rules are of this form, and have been elided. A call to a function f becomes a call to the function f_{ext}, where argument sizes are passed explicitly.

6.4.4 SOAC-related Rules

There are two issues to be dealt with when size-transforming SOACs. The first is that the type rules indicate that the outer dimension of all input arrays must be identical. We deal with this by using **reshape** to transform all input arrays to have the same outermost size as the first input array, which produces an error at run-time if the sizes do not match.

The second, and bigger problem with annotating SOACs is that, in Futhark, anonymous functions cannot be existentially quantified. Hence, before evaluating the SOAC, we must know the shape of its return value. For this section, we say that given an anonymous function λ, we can compute the function λ_{shape}, which returns the shape of the values that would normally be returned by λ.

Similarly to function calls, when transforming **map** λ a, there are two possible avenues of attack.

Pre-assertion First, calculate **map** λ_{shape} a, and assert, for each returned array, that all of its elements are identical. Put another way, this check in advance that the **map** expression results in a regular array. After this check, we know with certainty the shape of the values returned by the original map computation (and

$$
\begin{aligned}
&\mathcal{A}^{\text{exp}}_{\Sigma}(\textbf{map}\ \lambda\ \overline{a}^{(n)})) = \\
&\quad \textbf{let}\ s^2_1 \ldots s^{\text{d}(\tau_1)}_1 \ldots s^2_m, \ldots, s^{\text{d}(\tau_m)}_m\ =\ \lambda_{\text{shape}}\ \overline{a_i\,[\,0\,]}^{(n)} \\
&\quad \textbf{in map}\ \lambda_{\text{checking}}\ \overline{\text{shapeup}(a_i)}^{(n)} \\
&\text{where}\ \ (\overline{\tau}^{(m)}) \qquad = \text{Return type of } \lambda \\
&\qquad \text{shapeup}(a_i) \quad = \textbf{reshape}\ (\text{head}(\Sigma(a_1)) :: \text{tail}(\Sigma(a_i)))\ a_i \\
&\qquad r_i \qquad = s^2_i :: \ldots :: s^{\text{d}(\tau_i)}_i \\
&\qquad p_i \qquad = \text{tail}(\Sigma(a_i)) \\
&\qquad \lambda_{\text{checking}} \quad = \mathcal{A}^{\text{lam}}_{\Sigma}(\lambda, \overline{r}^{(m)}, \overline{p}^{(n)}).
\end{aligned}
$$

Figure 47: Transformation for **map** using the intra-assertion strategy.

that it will be regular), which we can then use to annotate the **map** computing the values. This strategy would require an extension to support an explicit **assert** expression.

Intra-assertion Alternatively, we can compute $\lambda_{\text{shape}}\ a[0]$, the shape of the first element of the result. Then, we modify λ to explicitly **reshape** its result to the same shape as that computed for the first element - we call this modified version $\lambda_{\text{checking}}$. This is the approach we used on Figure 40, and describe on Figure 47. First, the shape slice of λ is applied to the first element of the input, which results in one result for every dimension of every array-typed return value of λ. For brevity we have elided the branch guarding against the case where the input arrays are empty. This results in a prediction for the result size of λ, which is incorporated into constructing $\lambda_{\text{checking}}$, which explicitly **reshape**s its result to match the prediction.

Then we apply $\lambda_{\text{checking}}$ to the original input, although with every input array reshaped so that the outer dimension matches the outer dimension of the first input array, and the other dimensions unchanged. This is to respect the type rule that all input arrays to **map** (and the other SOACs) must have the same outer size.

The former approach is only efficient if λ_{shape} is efficient compared to λ - in practice, it must not contain any loops. The latter approach is limited in that it forces shape checking inside the value computation, which means that we do not know in advance whether the shape annotation is correct. However, in practice, the **reshape** operations often end up being statically simplified away. Hence, our compiler always applies the intra-assertion rule, with the expectation that a later optimisation will remove the assertions, if possible.

Similar approaches are used for the remaining SOACs. The cases for **reduce**, **scan**, **scatter**, and **stream_red** are simple because the correct return shape for

the anonymous functions can be deduced from the inputs to the SOAC. The case for **`stream_map`** is simple because the return shape must match the chunk size. These cases are all handled by inserting **`reshape`** expressions in the functions.

6.5 Related Work

An important piece of related work is the work on the FISh [Jay99] programming language, which uses partial evaluation and program specialization for resolving shape information at compile time. The semantics of FISh guarantees that this is possible. A similar approach is used in $\tilde{\text{F}}$ [Sha+17], with the specific motivation of being able to efficiently pre-allocate memory.

Futhark uses a similar strategy, but adds existential types to handle constructs such as **`filter`**, at the cost of no longer being able to fully resolve shape information statically in all cases.

Much work has also gone into investigating expressive type systems, based on dependent types, which allow for expressing more accurately the assumptions of higher-order operators for array operations [TC13; TG11; TG09]. Compared to the present work, such type systems may give the programmer certainty about particular execution strategies implemented by a backend compiler. The expressiveness, however, comes at a price. Advanced dependent type systems are often very difficult to program and modularity and reusability of library routines require the end programmer to grasp, in detail, the underlying, often complicated, type system. Computer algebra systems [Wat+90; Wat03] have also provided for a long time a compelling application of dependent types in order to express accurately the rich mathematical structure of applications, but inter-operating across such systems remains a significant challenge [Chi+04; OW05].

A somewhat orthogonal approach has been to extend the language operators so that size and bounds checking invariants always hold [ED14], the downside being that non-affine indexing might appear. The Futhark strategy is instead to rely on advanced program analysis and compilation techniques to implement a pay-as-you-go scheme for programming massively parallel architectures.

Another strand of related work is the work on effect systems for region-based memory management [Tof+04], in particular, the work on multiplicity inference and region representation analysis in terms of physical-size inference [Vej94; BTV96]. Whereas the goal of multiplicity inference is to determine an upper bound to the number of objects stored into a region at runtime, physical-size inference seeks to compute an upper bound to the number of bytes stored in a single write. Compared to the present work, multiplicity inference and physical-size inference are engineered to work well for common objects such as pairs and closures, but the techniques work less well with objects whose sizes are determined dynamically.

Chapter 7

Fusion

This chapter describes the approach taken by the Futhark compiler to perform loop fusion on SOACs. Our approach is capable of handling both *vertical fusion*, where two SOACs are in a producer-consumer relationship, and *horizontal fusion*, where two otherwise independent SOACs take the same array as input. As all other optimisations in the Futhark compiler, the approach is based on a syntactic rewriting of abstract syntax trees.

The core of the technique has been previously described in the author's master's thesis [Hen14]. The current thesis makes three new contributions:

1. A simple technique for integrating horizontal fusion in the existing framework. The previously published work did not support horizontal fusion at all.
2. Vertical fusion of **`map`** with **`scan`** and **`map`** with **`scatter`**.
3. Fusion rules for the streaming SOACs **`stream_red`** and **`stream_map`**.

However, before we can move on to the new contributions, we must establish the basic fusion algorithm. The algorithm identifies pairs of SOACs that can be fused. The rules by which we combine SOACs through fusion is called the *fusion algebra*. The fusion algebra used by the Futhark compiler has as its central goals to never duplicate computation, and never reduce available parallelism. This ensures that the asymptotics of the program under optimisation are not affected. Instead, the goal of fusion is to eliminate the overhead of storing intermediate results in memory.

Most fusion algorithms in the literature are unable to handle fusion across **`zip`**/**`unzip`**, and more generally the case where the output of a producer is used by several consumers. The algorithm used by the Futhark compiler is capable of fusing such cases if this is possible without duplicating computation, as demonstrated on Figure 48. The linchpin of this capability is our choice of internal representation, in which arrays of tuples (and therefore **`zip`**/**`unzip`**) do not occur.

```
let b = map (+1) a
let c = map (+2) b
let d = map (+3) b
in map (+) c d
```

(a) Before fusion.

```
map (λx →
      let b = x + 1
      let c = b + 2
      let d = c + 3
      in c + d)
    a
```

(b) After fusion.

Figure 48: Fusing multiple consumers without duplicating computation.

This chapter covers two main themes: Section 7.2 describes our aggressive fusion algorithm, which in particular supports fusion for producers whose results are used by multiple consumers, without duplicating computation. Section 7.3 describes the rewrite rules used to fuse two SOACs. However, first we define an extended set of SOACs with better fusion properties than the SOACs used in previous chapters.

7.1 New SOACs

The SOACs we have used so far are a close match to the SOACs of the source Futhark language. However, they do not permit a fusion algebra as rich as we desire. Using the previously shown SOACs, there is not even a way to fuse a composition of **`map`** and **`reduce`**. This section will introduce a new set of SOACs that, while perhaps not as aesthetically pleasing and orthogonal as those in the source language, have two important qualities:

1. Their fusion properties are much more powerful, permitting for example both vertical and horizontal fusion of **`map`** with **`reduce`** or **`scan`**.
2. They permit an efficient mapping to parallel code. This requirement prevents us from defining overly complicated SOACS that fuse with everything, but no longer have parallel semantics. This issue is examined in more depth in Chapter 8.

The types of the new SOACs are shown in Figure 49. An individual description now follows, which also shows how instances of the previous SOACs are mapped to the new ones. It may prove useful to refer to the types while reading the descriptions.

`map` is unchanged from before.

`scatter` now takes a functional argument, which maps n input arrays to m pairs of indexes and values, which are written to the m destination arrays, if the indexes are in bounds. What we previously wrote as

```
scatter dest is vs
```

we will now write as

```
scatter (λi v → (i,v)) (dest) is vs
```

The `dest` is in parentheses to distinguish the destination arrays from the input arrays.

redomap is a composition of **map** and **reduce** that permits a particularly efficient implementation. It corresponds to a variation of the redomap operator from Section 3.2.2. A **redomap** has two functional arguments: an associative reduction operator and a fold function. The fold function takes as parameters a current *accumulator*, and an element of the input. The **redomap** fold function returns $m + l$ values. The first m are called the *reduced results*, and are passed on to the reduction operator, while the remaining l *mapped results* are simply returned as an array in the final result. We shall soon see how this enables horizontal fusion. A source-language **reduce** is merely a special case of **redomap**:

```
reduce f x a = redomap f f x a
```

The way we construct the fold function in **redomap** ensures that the mapped results do not depend on the accumulator. It trivially holds when a **redomap** is first constructed from a **reduce** (because there are no mapped results), and is maintained by the fusion algebra. This is crucial for generating parallel code from **redomap**. As an example of **redomap** fusion, consider the following expression fragment:

```
let (ys: [n]i32) = map (λx → 0 - x) xs
let (zs: [n]i32) = map (λx → x + 1) xs
let (sum: i32)   = reduce (+) (0) xs
in (ys, zs, sum)
```

This can be fused into a single **redomap**, as follows:

```
let (sum: i32) (ys: [n]i32) (zs: [n]i32) =
  redomap (+)
          (λx →
            (x,      -- reduced result
             0 - x,  -- mapped result
             x + 1)  -- mapped result
          )
          (0)
          xs
in (ys, zs, sum)
```

This fusion operation would not be possible without the notion of mapped results.

scanomap is similar to **redomap**, but performs a scan instead of a reduction. The following equivalence holds:

scan f x a = **scanomap** f f x a

stream_par functions as a unification of the **stream_map** and **stream_red** constructs from the source language. As with **redomap**, the chunk operator returns $m + l$ values, of which the first m take part in reduction, and the latter l are simply returned. One restriction is that the latter l values must all be arrays of size x, where x is the size of the chunk of the input given to the operator.

stream_seq is a SOAC used to fuse otherwise infusible cases, without losing potential parallelism, by performing what is semantically a fold over chunks of the input array. The chunk operator takes the current values of the accumulator, as well as chunks of the n input arrays, and returns new values for the accumulator. We can recover all parallelism in the chunk operator by applying it to "chunks" containing the full input arrays, or fully sequentialise by adding an outer sequential loop and setting the chunk size to 1. This aids in efficient sequentialisation. We discuss **stream_seq** in greater detail in Section 7.3.2.

We exclude **filter** from our discussion of fusion. While **filter** does have useful fusion properties (in particular with **reduce**/**redomap**), the current implementation in the Futhark compiler implements **filter** by a decomposition into **scan** and **scatter**.

7.2 The Overall Fusion Algorithm

While the foundations of the fusion algorithm were developed prior to this thesis [HO13], we will summarise the basic technique. The following is adapted from the author's master's thesis [Hen14].

op		TySch(*op*)
map	:	$\forall d\bar{\alpha}^{(n)}\bar{\beta}^{(m)}.(\overline{\alpha_i}^{(n)} \to (\bar{\beta_i}^{(m)})) \to \overline{[d]\alpha_i}^{(n)} \to (\overline{[d]\beta_i}^{(m)})$
scatter	:	$\forall d\overline{x}^{(m)}\bar{\alpha}^{(n)}\bar{\beta}^{(m)}.$ $(\overline{\alpha}^{(n)} \to (\overline{\texttt{i32}, \beta_1}^{(m)})) \to (\overline{[x_i]\beta_i}^{(m)}) \to (\overline{[d]\alpha_i}^{(n)}) \to (\overline{[x_i]\beta_i}^{(m)})$
redomap	:	$\forall d\bar{\alpha}^{(m)}\bar{\beta}^{(n)}\overline{\gamma}^{(l)}.$ $(\overline{\alpha}^{(m)} \to \overline{\alpha}^{(m)} \to (\bar{\alpha}^{(m)})) \to (\overline{\alpha}^{(m)} \to \overline{\beta}^{(n)} \to (\bar{\alpha}^{(m)}, \bar{\gamma}^{(l)}))$ $\to (\overline{\alpha}^{(m)}) \to \overline{[d]\beta_i}^{(n)} \to (\overline{\alpha}^{(m)}, \overline{[d]\gamma_i}^{(l)})$
scanomap	:	$\forall d\bar{\alpha}^{(m)}\bar{\beta}^{(n)}\overline{\gamma}^{(l)}.$ $(\overline{\alpha}^{(m)} \to \overline{\alpha}^{(m)} \to (\bar{\alpha}^{(m)})) \to (\overline{\alpha}^{(m)} \to \overline{\beta}^{(n)} \to (\bar{\alpha}^{(m)}, \bar{\gamma}^{(l)}))$ $\to (\overline{\alpha}^{(m)}) \to \overline{[d]\beta_i}^{(n)} \to (\overline{[d]\alpha}^{(m)}, \overline{[d]\gamma_i}^{(l)})$
stream_par	:	$\forall dx\bar{\alpha}^{(m)}\bar{\beta}^{(n)}\overline{\gamma}^{(l)}.$ $(\overline{\alpha}^{(m)} \to \overline{\alpha}^{(m)} \to (\overline{\alpha}^{(m)}))$ $\to ((x : \texttt{i32}) \to \overline{[x]\beta_i}^{(n)} \to (\bar{\alpha}^{(m)}, \overline{[x]\gamma_i}^{(l)}))$ $\to \overline{[d]\beta_i}^{(n)} \to (\overline{\alpha}^{(m)}, \overline{[d]\gamma_i}^{(l)})$
stream_seq	:	$\forall dx\bar{\alpha}^{(m)}\bar{\beta}^{(n)}\overline{\gamma}^{(l)}.$ $((x : \texttt{i32}) \to \overline{\alpha}^{(m)} \to \overline{[x]\beta_i}^{(m)} \to (\bar{\beta}^{(m)}, \overline{[x]\gamma}^{(l)}))$ $\to (\overline{\alpha}^{(m)}) \to \overline{[d]\beta_i}^{(n)} \to (\overline{\alpha}^{(m)})$

Figure 49: Extended SOACs used for fusion and later stages of the compiler.

The entire algorithm consists of two distinct stages:

1. Traverse the program, collecting SOAC-expressions and fusing producers with consumers where possible. The end result is a mapping from SOACs in the original program, to replacement SOAC expressions (the result of fusion operations). This is called the *gathering* phase.

2. Traverse the program again, using the result of the gathering phase to replace SOAC expressions with their fully fused forms. This may lead to dead code, as the output variables of producers that have been fused with their consumers are no longer used. These can be removed using standard dead code removal.

Futhark, as a block-structured language, is suited to region-based analysis, and the fusion algorithm is indeed designed as a reduction of a dataflow graph. Our structural analysis is inspired by the T_1-T_2-reduction [Aho+07], which says that a flow graph is reducible if it can be reduced to a single node by the following two transformations:

T_1: Remove an edge from a node to itself.

T_2: Combine two nodes m and n, where m is the single predecessor of n, and n is not the entry of the flow graph.

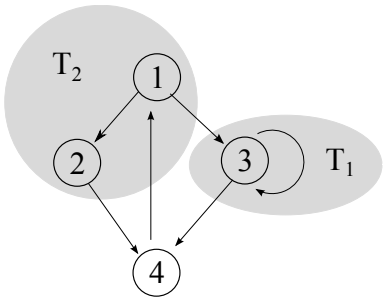


Figure 50: T_1-T_2-reduction

Figure 50 shows a small flow graph and highlights instances where the two reductions could apply. The overall idea is to construct a flow graph of the Futhark program, reduce it to a single point, and at each reduction step combine the information stored at the nodes being combined.

By construction, Futhark always produces a reducible graph. Each node corresponds to an expression, with the successors of the node being its subexpressions. This means that we can implement the reduction simply as a bottom-up traversal of the Futhark syntax tree.

Figure 51 depicts the intuitive idea on which our fusion transformation is based. The top-left figure shows the dependency graph of a simple program, where an arrow points from the consumer to the producer.

The main point is that all SOACs that appear inside the dashed circle can be fused without duplicating any computation, even if several of the to-be-fused arrays are used in different SOACs. For example, `y1` is used to compute both `(z1,z2)` and `(q1,q)`[1]. This is accomplished by means of T_2 reduction on the dependency graph: The rightmost child, i.e., **`map`** `g ...`, of the root SOAC (**`map`** `f1 ...`) has only one incoming edge, hence it can be fused. This is achieved by:

1. Replacing in the root SOAC the child's output with the child's input arrays
2. Inserting a call to the child's function in the root's function, which computes the per-element output of the child,
3. Removing duplicate input arrays of the resulting SOAC.

This is the procedure for fusing two **`map`** SOACs. Section 7.3 gives the rules used for other cases.

The top-right part of Figure 51 shows the result of the first fusion, after slight simplification of the lambda bodies via copy propagation. In the new graph, the leftmost child of the root, i.e., the one computing `(z1,z2)`, has only one incoming

[1] Note also that not all input arrays of a SOAC need be produced by the same SOAC.

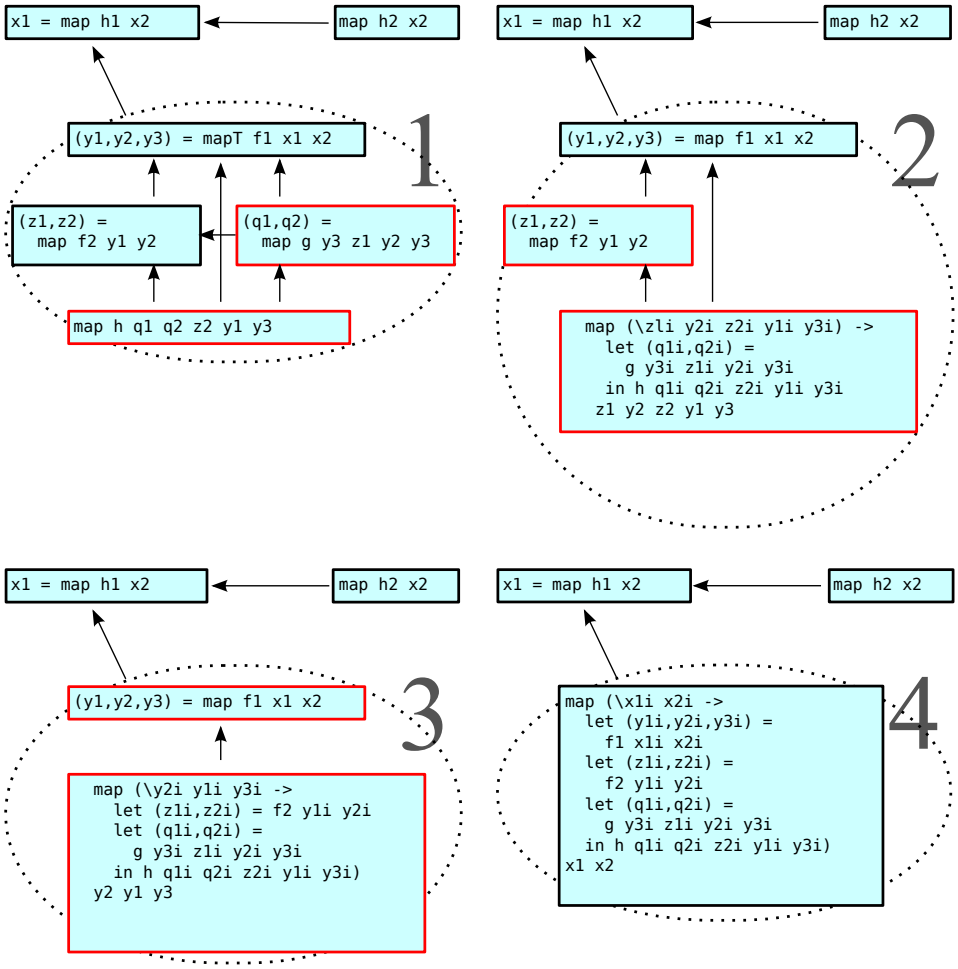
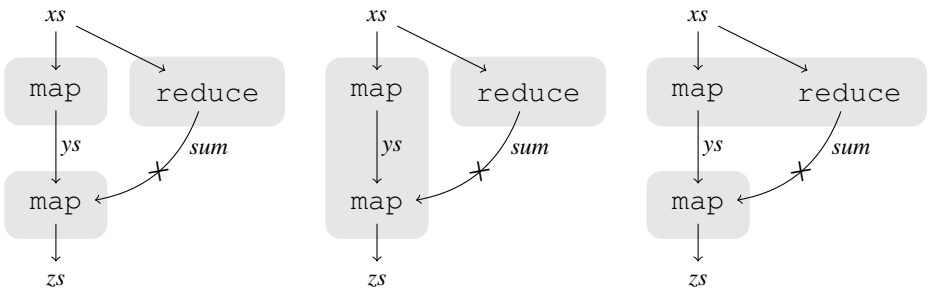


Figure 51: Fusion by T_2-transformation on the dependency graph

edge and can be fused. The resulting graph, shown in the bottom-left figure can be fused again resulting in the bottom-right graph of Figure 51. At this point no further T_2 reduction is possible, because the SOAC computing `x1` has two incoming edges. This example demonstrates a key benefit of removing **`zip`/`unzip`** and using our tupleless SOACs representation: there are no intermediate nodes in the data-dependency graph between fusible producer and consumer.

7.2.1 Optimality of Fusion

With a sufficiently rich fusion algebra it is sometimes the case that the data dependency graph can be reduced in multiple different ways. Consider the following example from [RLK14].



(a) No fusion.

(b) Vertical fusion of the two **map** operations.

(c) Horizontal fusion of **map** and **reduce** into **redomap**.

Figure 52: Three fusion alternatives for the example program. Figure taken from [Dyb17] with permission.

```
let ys = map (λx → x + 1.0) xs
let sum = reduce (+) (0.0) xs
let zs = map (λy → y / sum) ys
in zs
```

There are three mutually incompatible ways to fuse in this expression.

- No fusion. (Figure 52a).
- Fuse the two **map**s together, yielding a single **map** that has a dependency on the **reduce** through `sum` (Figure 52b).
- Horizontally fuse the first **map** with the **reduce**, yielding a **redomap**. This **redomap** cannot be fused with the final **map** (Figure 52c).

The crossed-out edge indicates an incontractable edge, because while there is a data dependency between the **reduce** and the final **map**, it is not of a form that permits fusion.

Looking at the expression in isolation, we might be able to use heuristics based on hardware propertiers to determine whether it is better to fuse one way or the other. However, for larger expressions the choice made might affect fusibility of other SOACs that are in a producer- or consumer-relationship with those fused. Determining optimal fusion is equivalent to graph clustering, an NP-hard problem. However, practical solutions based on integer linear programming [MS97] or graph partitioning [Kri+16] have been developed. The Futhark compiler presently uses a *greedy* approach: the first succesful candidate for fusion is used (where the iteration order is essentially just the bottom-up traversal of the AST). We have not yet in practice observed cases where the greedy approach leads to suboptimal code, but if such cases were to become prevalent, the techniques mentioned above could readily be adopted.

7.2.2 Invalid fusion

We must be careful not to violate the rules on in-place updates (Section 5.3.2) when performing fusion. For example, consider the following program.

```
let b = map f a
let c = a with [i] ← x
in map g b
```

Without the constraints imposed upon us by the semantics of in-place modification, we could fuse to the following program.

```
let c = a with [i] ← x
in map (g ∘ f) a
```

However, this results in a violation of Uniqueness Rule 1, because `a` is now used after it has been consumed, and the resulting program is thus invalid. In general, we must track the possible execution paths from the producer-SOAC to the consumer-SOAC, and only fuse if none of the inputs of the producer have been consumed (in the uniqueness type sense of the word) by an in-place update or function call on any possible execution paths. This is easier than it may appear at first glance, as the fusion algorithm will anyway only fuse when the consumer is within the lexical scope of the producer.

Another, subtler, issue is that a fused SOAC may consume more of its input than the original constituent SOAC. Consider the following expression.

```
let b = map (λx → x) a
in map g b
```

Suppose that the function `g` consumes its input. This means that in the expression as a whole, the array `b` is being consumed, but `a` is not (by the aliasing rules, the array produced by a **map** has no aliases). After **map**-**map** fusion (and copy propagation), we obtain

```
map (λx → g x) a
```

This expression *does* consume `a`. While this example is contrived, it is not impossible for `f` to be a function whose output aliases its input, which will give the same problem.

The solution is to track which inputs are consumed, for each SOAC that contributed to the final SOAC produced at the end of fusion. For each array that the final SOAC consumes, yet which were not consumed by any of the original SOACs, we add a `copy`. In the example above, this would correspond to the following.

```
let a' = copy a
in map (λx → g x) a'
```

This ensures that the result of fusion never consumes more than the original expression(s). For this example, it also makes fusion ineffective, as the **copy** is just as slow as an identity **map**. However, this is the worst-case situation: at most we add one **copy** per SOAC input fused, which will never be worse than not fusing in the first place, but also might not be an improvement.

7.2.3 When to fuse

Even when fusion is possible, it may not be beneficial, and may be harmful to overall performance. Two such cases are discussed below.

Fusion may duplicate computation.

In the program

```
let xs = map f as
let ys = map g xs
let zs = map h xs
in (ys,zs)
```

vertically fusing the `x`-producer with the two consumers will double the number of calls to the function `f`, which might be expensive. The implementation in the Futhark compiler will currently only fuse if absolutely no computation is duplicated, although this is likely too conservative. Duplicating cheap work, for example functions that use only primitive operations on scalars, is probably not harmful to overall performance, although we have not investigated this fully.

In general, the tradeoff between duplicating computation and increasing communication is not an easy problem to solve. On a GPU, Accessing global memory can be more than a hundred times slower than accessing local (register) memory, and hence duplicating computation may in some cases be preferable.

The fusion algorithm used by the Futhark compiler is careful never to duplicate computation. However, some simplification rules may duplicate small amounts of scalar computation.

Note that the expression above is only a problem when considering just vertical fusion; using horizontal fusion we can fuse this into a single **map** (see Section 7.3.3 where we use just this example).

Fusion may reduce memory locality.
Consider a simple case of fusing

$$\texttt{map}\ f \circ \texttt{map}\ g$$

When g is executed for an element of the input array, neighboring elements may be put into the cache, making them faster to access. This exhibits good data locality. In contrast, the composed function $f \circ g$ will perform more work after accessing a given input element, increasing the risk that the input array may be evicted from the cache before the next element is to be used. On GPUs, there is the added risk of the kernel function exercising additional register pressure, which may reduce hardware occupancy (thus reducing latency hiding) by having fewer computational cores active. In this case, it may be better to execute each of the two `map`s as separate kernels.

The fusion algorithm used by the Futhark compiler is not concerned with these issues. The risk of cache eviction is not very relevant on GPUs, and determining the proper size of GPU kernels is a job better left for kernel extraction (Chapter 8).

7.3 Fusion Rules

The rules of the fusion algebra describe the fusion of two SOACs with associated patterns, termed A and B, where A produces some arrays that are inputs to B. Syntactically, we write the SOACs as a **`let`**-binding without the **`in`** part, as **`let`** p `=` e. This is important because fusion not only rewrites the SOAC itself, but also the pattern to which it is bound. A crucial property is that, when fusing SOACs A and B, the resulting SOAC C binds (at least) the same names as A and C. This permits us to remove A entirely, and replace B with C. It is likely that some of these names will be dead after fusion, but they will be removed by subsequent simplification, not by the fusion algorithm itself.

For simplicity, when fusing A into B, we will assume that the inputs to B have been arranged so that those arrays that are produced by A come first. This causes no loss of generality, as this rearranging is always possible. We commit some abuse of notation when describing the composition of the lambda functions, as we invoke lambdas as if they were functions. This is not strictly permitted by the grammar we use for the core language, but it avoids a significant amount of tedious bookkeeping in the rules.

We do not describe explicitly rules for all compositions of SOACs, even those that are in principle fusible. Since any **`map`** can be transformed into a **`scanomap`** or **`redomap`** by the equivalence on Figure 53, and any **`redomap`** can be transformed into a **`stream_par`** by the equivalence on Figure 54, we elide those cases that can be handled by first transforming the SOAC into a more general form. This does for

Given a SOAC

$$\texttt{let } \overline{ys} \texttt{ = map } f\ (\overline{v})\ \overline{xs}$$

the following SOACs are both equivalent:

$$\texttt{let } \overline{ys} \texttt{ = scanomap nil } f\ ()\ \overline{xs}$$

and

$$\texttt{let } \overline{ys} \texttt{ = redomap nil } f\ ()\ \overline{xs}$$

where **nil** is the anonymous function that accepts zero arguments and returns zero values (which would otherwise be written $(\lambda\rightarrow)$).

Figure 53: Transforming a **map** to a **scanomap** or **redomap**.

Given a SOAC

$$\texttt{let } \overline{y}\ \overline{ys} \texttt{ = redomap } \oplus\ f\ (\overline{vs})\ \overline{xs}$$

the following SOAC is equivalent:

$$\texttt{let } \overline{y}\ \overline{ys} \texttt{ = stream\_par } \oplus\ h\ (\overline{vs})\ \overline{xs}$$

where

$$\begin{aligned} h = \lambda c\ \overline{xs'} \rightarrow\ & \\ & \texttt{let } \overline{y'}\ \overline{ys'} = \texttt{redomap } \oplus\ f\ (\overline{vs})\ \overline{xs'} \\ & \texttt{in } (\overline{y'}, \overline{ys'}) \end{aligned}$$

Figure 54: Transforming a **redomap** to a **stream_par**. Note that the function of the produced **stream_par** itself contains a **redomap**—an implementation must take care not to apply this transformation when unnecessary for fusion, or risk nontermination. This transformation relies on the guarantee that the mapped result of **redomap** cannot depend on the accumulator.

example mean that by the rules, **map**-**map** fusion produces a **redomap**, not a **map**, but a **map** can be recovered by exploiting the equivalence in the opposite direction. Reverting back to less general constructs is necessary because the moderate flattening algorithm of Chapter 8 depends crucially on being able to recognise **map** nestings, and can also benefit from recognising simple cases of **reduce** and **scan**.

Recovering a **redomap** from a **stream_par** is more difficult, and is not attempted by the Futhark compiler. In general, **stream_par**s only occur if the original source program contained a **stream_red**. All SOACs can be transformed into **stream_seq**; the advantage of which we will discuss in Section 7.3.2.

Not all pairs of SOACs are covered by the following fusion rules, even when permitting transformation into **stream_par**. These cases are handled by transforming

the remaining SOACs involved into **stream_seq**, as discussed in Section 7.3.2.

7.3.1 List of Fusion Rules

map-scatter

SOAC A :	$\texttt{let}\ \overline{ys} = \texttt{map}\ f\ \overline{xs_A}$
SOAC B :	$\texttt{let}\ \overline{zs} = \texttt{scatter}\ g\ (\overline{vs})\ \overline{ys}\ \overline{xs_B}$
Fuses to SOAC C :	$\texttt{let}\ \overline{zs} = \texttt{scatter}\ h\ (\overline{vs})\ \overline{xs_A}\ \overline{xs_B}$

$$
\begin{aligned}
\text{where } h = \lambda \overline{x}\ \overline{y} \rightarrow \\
&\texttt{let}\ \overline{y} = f\ \overline{x} \\
&\texttt{let}\ \overline{z} = g\ \overline{y} \\
&\texttt{in}\ \overline{z}
\end{aligned}
$$

Note that in contrast to other fusion rules, all outputs of SOAC A must be used by SOAC B. In the future, it is likely that **scatter** will gain mapped results, like **redomap**, to enable greater fusibility.

redomap-redomap

SOAC A:	$\texttt{let}\ \overline{y_r}\ \overline{ys_m}\ \overline{ys_s} = \texttt{redomap}\ \oplus\ f\ (\overline{v_A})\ \overline{xs_A}$
SOAC B:	$\texttt{let}\ \overline{z_r}\ \overline{zs_s} = \texttt{redomap}\ \otimes\ g\ (\overline{v_B})\ \overline{ys_m}\ \overline{xs_B}$
Fuses to SOAC C:	$\texttt{let}\ \overline{y_r}\ \overline{z_r}\ \overline{zs_s}\ \overline{ys_m}\ \overline{ys_s} = \texttt{redomap}\ \odot\ h\ (\overline{v_A}, \overline{v_B})\ \overline{xs_A}\ \overline{xs_B}$

$$
\begin{aligned}
\text{where } h = \lambda \overline{a_A}\ \overline{a_B}\ \overline{x}\ \overline{y} \rightarrow \\
&\texttt{let}\ \overline{c_A}\ \overline{y_m}\ \overline{y_s} = f\ \overline{a_A}\ \overline{x} \\
&\texttt{let}\ \overline{c_B}\ \overline{z} = g\ \overline{a_B}\ \overline{y_m} \\
&\texttt{in}\ (\overline{c_A}, \overline{c_B}, \overline{z}, \overline{y_m}, \overline{y_s}) \\
\odot = \lambda \overline{a_A}\ \overline{a_B}\ \overline{b_A}\ \overline{b_B} \rightarrow \\
&\texttt{let}\ \overline{c_A} = \oplus\ \overline{a_A}\ \overline{b_A} \\
&\texttt{let}\ \overline{c_B} = \otimes\ \overline{a_B}\ \overline{b_B} \\
&\texttt{in}\ (\overline{c_A}, \overline{c_B})
\end{aligned}
$$

The outputs of SOAC A are divided into three parts: y_r are the reduction results. These are not used by SOAC B at all, or the fusion algorithm would not have attempted fusion in the first place (there would be an incontractable edge in the graph). The map results are divided into ys_m, which are input to SOAC B, and

ys_s, which are not. SOAC C must still produce both of these, even though it is likely that ys_m will not be used afterwards.

Note also how we combine the reduction operators $\oplus$ and $\otimes$ to produce $\odot$. We do this through no particular cleverness; as we simply apply $\oplus$ and $\otimes$ to those parameters that correspond to the original values from SOAC A and B. This operation carries a risk, as it conceptually increases the size of the values involved in the reduction. The implementation of reductions shown in Section 4.5 uses GPU local memory to communicate between threads during the reduction, and for sufficiently large values we may eventually run out of local memory. In such cases we either have to communicate between threads in (slow) global memory, or split apart the reduction into multiple independent passes. The Futhark compiler currently does not address this problem. Since fusion is performed statically (as opposed to dynamically; see Section 7.4 for alternatives), this risk is not dataset-dependent, but only proportional to the number of independent reductions present in the program being compiled. We have not yet observed this problem in practice.

`scanomap-scanomap`

Similar to **`redomap-scanomap`**.

`stream_par-stream_par`

SOAC A:	**let** $\overline{y_r}\ \overline{ys_m}\ \overline{ys_s}$	= **stream_par** $\oplus\ f\ \overline{xs_A}$
SOAC B:	**let** $\overline{z_r}\ \overline{zs}$	= **stream_par** $\otimes\ g\ \overline{ys_m}\ \overline{xs_B}$
Fuses to SOAC C:	**let** $\overline{z_r}\ \overline{y_r}\ \overline{zs}\ \overline{ys_m}\ \overline{ys_s}$	= **stream_par** $\odot\ h\ \overline{xs_A}\ \overline{xs_B}$

$$
\begin{aligned}
\text{where } h = \lambda\ c\ \overline{xs'_A}\ \overline{xs'_B} \rightarrow\ & \\
& \textbf{let}\ \overline{y_A}\ \overline{ys'_m}\ \overline{ys'_s} = f\ c\ \overline{xs'_A} \\
& \textbf{let}\ \overline{z_B}\ \overline{zs'} = g\ c\ \overline{ys'_m}\ \overline{xs'_B} \\
& \textbf{in}\ (\overline{z_B}, \overline{y_A}, \overline{zs'}, \overline{ys'_m}, \overline{ys'_s}) \\
\odot = \lambda \overline{a_A}\ \overline{a_B}\ \overline{b_A}\ \overline{b_B} \rightarrow\ & \\
& \textbf{let}\ \overline{c_A} = \oplus\ \overline{a_A}\ \overline{b_A} \\
& \textbf{let}\ \overline{c_B} = \otimes\ \overline{a_B}\ \overline{b_B} \\
& \textbf{in}\ (\overline{c_A}, \overline{c_B})
\end{aligned}
$$

Fusion of **`stream_par`** is very similar to **`redomap`** fusion, and carries the same potential problem regarding the combined reduction operator.

stream_seq-stream_seq

SOAC A:	**let** $\overline{y_r}\ \overline{ys_m}\ \overline{ys_s}$	= **stream_seq** $f\ (\overline{v}_A)\ \overline{xs_A}$
SOAC B:	**let** $\overline{z_r}\ \overline{zs}$	= **stream_seq** $g\ \overline{ys_m}\ (\overline{v}_B)\ \overline{xs_B}$
Fuses to SOAC B:	**let** $\overline{y_r}\ \overline{z_r}\ \overline{zs}\ \overline{ys_m}\ \overline{ys_s}$	= **stream_seq** $h\ \overline{ys_m}\ (\overline{v}_B)\ \overline{xs_B}$

$$
\begin{aligned}
\text{where } h = \lambda\ c\ \overline{a_A}\ \overline{a_B}\ \overline{xs'_A}\ \overline{xs'_B} \rightarrow\ & \\
& \textbf{let}\ \overline{y_A}\ \overline{ys'_m}\ \overline{ys'_s}\ = f\ c\ \overline{a_A}\ \overline{xs'_A} \\
& \textbf{let}\ \overline{z_B}\ \overline{zs'} = g\ c\ \overline{a_B}\ \overline{ys'_m}\ \overline{xs'_B} \\
& \textbf{in}\ (\overline{z_B}, \overline{y_A}, \overline{zs'}, \overline{ys'_m}, \overline{ys'_s})
\end{aligned}
$$

7.3.2 Sequential Fusion via `stream_seq`

There are some producer-consumer relationships that cannot be fused by the above rules. For example, a **scanomap** whose output is used as inputs to a **map**. This is because there is no SOAC that is able to describe the resulting composition without duplicating computation. We could fuse and produce a sequential **do**-loop as a result, but this would lose parallelism, which is not acceptable in general. However, if the **scanomap**-**map** composition occurs at a place that will eventually be turned into sequential code by the moderate flattening algorithm described in Chapter 8, then such fusion would be desirable. But since fusion occurs at a stage where it is not yet known which parts of the program will be executed in parallel, and which will be sequential, we have a conundrum.

The solution is to fuse to a SOAC that permits the *recovery* of all original parallelism, yet can also be turned into efficient sequential code. We use **stream_seq** for this purpose. Any SOAC can be transformed into a **stream_seq**, so by transforming two SOACs that cannot otherwise be fused by the fusion algebra intoto **stream_seq**s, then using the rule for **stream_seq** fusion, we can in effect fuse any two SOACs.

For example, the transformation from a **scanomap** into **stream_seq** is shown on Figure 55. The idea is for each chunk to perform a **scanomap**, then add to the result the carry produced by processing the previous chunk. The carry for a chunk corresponds to the last element of a scanned chunk. This works only because we are guaranteed that the scan operator $\oplus$ is associative. The initial value of the carry is the neutral element for $\oplus$.

The utility of **stream_seq** is that we can recover the parallelism of the original formulation simply by setting the chunk size to the full size of the input array, resulting in just one chunk. While the transformation on Figure 55 has introduced additional **map**s, these do not affect how much parallelism is available. Furthermore,

Given a SOAC

$$\textbf{let } \overline{ys} \textbf{ = scanomap } \oplus\ f\ (\overline{v})\ xs$$

the following SOAC is equivalent:

$$\textbf{let } \overline{d}\ \overline{ys} \textbf{ = stream\_seq } \oplus\ g\ (\overline{v})\ xs$$

where $\overline{d}$ are the unused final carry values and

$$\begin{aligned}
g = \lambda\ c\ \overline{a}\ \overline{xs'} \rightarrow\ & \\
& \textbf{let } \overline{xs''} \textbf{ = scanomap } \oplus\ f\ (\overline{v})\ xs' \\
& \textbf{let } ys'_1 \textbf{ = map } (\oplus\ a_1)\ xs''_1 \\
& \vdots \\
& \textbf{let } ys'_n \textbf{ = map } (\oplus\ a_n)\ xs''_n \\
& \textbf{let } a'_1 \textbf{ = } ys'_1[c-1] \\
& \vdots \\
& \textbf{let } a'_n \textbf{ = } ys'_n[c-1] \\
& \textbf{in } (\overline{a'}, \overline{ys'})
\end{aligned}$$

Figure 55: Transforming a **scanomap** to a **stream_seq**. We assume here that chunks may never be empty. Other SOACs can be transformed using a similar technique.

since the carry will be set to the neutral element, frequently a constant, basic simplification may be able to remove the **map**s entirely.

As an example, consider the following expression fragment, where we suppose the size of `xs` is given by `n`:

```
let (ys: [n]i32) = scanomap (+) (+) (0) xs
let (zs: [n]i32) = map (+2) ys
```

The above can be transformed to the following **stream_seq**s:

```
let (unused: i32) (ys: [n]i32) =
  stream_seq (λ(c: i32) (a: i32) (xs': [c]i32)
                : (i32, [c]i32) →
                let xs'' = scanomap (+) (+) 0 xs'
                let ys' = map (+) a xs''
                let a' = ys[c-1]
                in (a', ys'))
             (0)
             xs
let (zs: [n]i32) =
  stream_seq (λ(c: i32) (ys': [c]i32): [c]i32 →
                let zs' = map (+2) ys'
                in zs')
             ()
             ys
```

Note that we use mapped results to return the chunks making up the `ys` array, and discard the final accumulator. The two **`stream_seq`**s can be fused as follows:

```
let (ys: [n]i32) (zs: [n]i32) =
  stream_seq (λ(c: i32) (a: i32) (xs': [c]i32): i32 →
                let xs'' = scanomap (+) (+) 0 xs'
                let ys' = map (+) a xs''
                let a' = ys[c-1]
                let zs' = map (+2) ys'
                in (a', ys', zs'))
             (0)
             xs
```

We can recover the original parallelism simply by inlining the body of the anonymous function, preceded by explicit bindings of `c`, `a`, and `xs'` to values corresponding to a single chunk of size `n` covering all of `xs`, and followed by a binding of `ys` to the result:

```
let c = n
let a = 0
let (xs': [c]i32) = reshape (c) xs
let (xs'': [c]i32) = scanomap (+) (+) (0) xs'
let ys' = map (+) a xs''
let a' = ys[c-1]
let zs' = map (+2) ys'
let (ys: [n]i32) = reshape (n) ys'
let (zs: [n]i32) = reshape (n) zs'
```

Note the need for **reshape**s to make the size-dependent typing work out. An application of copy propagation, constant folding, and other straightforward simplifications, in particular removing identity reshapes, then recovers the original expression fragment.

On the other hand, we can also transform the above **stream_seq** into a sequential loop by forcing the chunk size to unit, and inserting an enclosing sequential **do**-loop:

```
let ys_blank = replicate n 0
let zs_blank = replicate n 0
let (ys, zs) =
  loop (a ys_out zs_out) =
       (0, ys_blank, zs_blank) for i < n do
    let c = 1
    let xs' = xs[i:i+c]
    let xs'' = scanomap (+) (+) 0 xs'
    let ys' = map (+) a xs''
    let a' = ys[c-1]
    let zs' = map (+2) ys'
    let ys_out' = ys_out with [i:i+c] ← ys'
    let zs_out' = zs_out with [i:i+c] ← zs'
    in (a', ys_out', zs_out)
```

(We cheat a bit notation-wise with the construction of such arrays as `xs'` and `ys_out'`—according to the grammar used thus far, indexes can only produce single elements, but here we perform an entire slice, and likewise perform an in-place update on an entire range of an array. These can be turned into **do**-loops if desirable.)

Assuming straightforward simplification rules, including removing SOACs on single-element input arrays, as well as unused **loop** results, we obtain the following:

```
let ys_blank = ...
let zs_blank = ...
let zs =
  loop (a, ys_out) = (0, ys_blank) for i < n do
    let x = xs[i]
    let y = a + x
    let z = 1 + y
    let ys_out' = ys_out with [i] ← y
    let zs_out' = zs_out with [i] ← z
    in (y, ys_out' zs_out')
```

This loop corresponds to a sequential fusion of the original **scanomap-map** composition, with no intermediate arrays.

7.3.3 Horizontal Fusion

Horizontal fusion can be treated as a special case of vertical fusion, with the same fusion rules. We simply consider the set of names in the producer/consumer relationship to be empty, and only fuse with respect to the unconsumed names. For example, for the **map**–**map** fusion rule, we let $\overline{ys_m}$ be empty. Two SOACs can be horizontally fused if their inputs have the same outer size, there is no data dependencies between them, and there are no uniqueness issues that would prevent the result of fusion from being safe. This is integrated easily with the fusion algorithm simply by adding edges in the graphs between SOACs that have the same outer size, and are not separated by being in different **loop** or **if** expressions.

While horizontal fusion provides a minor optimisation in reducing array traversals, its major purpose is as an enabler of vertical fusion. For example, consider the previous example of an expression that cannot be fused without duplicating computation, using just vertical fusion:

```
let xs = map f as
let ys = map g xs
let zs = map h xs
in (ys,zs)
```

Using horizontal fusion, we can fuse the expressions producing `ys` and `zs` as follows:

```
let xs = map f as
let (ys, zs) = map (λx → (g x, h x)) xs
in (y,z)
```

Which then permits vertical fusion of the expression producing `xs`:

```
let (ys, zs) = map (λa → let x = f a
                         in (g x, h x))
                   as
in (y,z)
```

7.4 Related Work

Loop fusion is an old technique, dating back at least to the seventies [Che77], with loop fusion in a parallel setting being covered in [MP90]. In work on imperative languages, the term "fusion" typically refers to horizontal, not vertical fusion. Most functional languages primarily make use of vertical fusion, because the rewrite rules are more local. Single Assignment C [GS06] (SaC) is, however, a prominent example of a functional language that incorporates horizontal fusion.

Data-Parallel Haskell (DPH) [Cha+07] makes use of aggressive inlining and rewrite rules to perform fusion, including expressing array operations in terms of streams [CSL07], which have previously been shown to be easily fusible. While DPH obtains good results, this form of rewrite rules are quite limited—they are an inherently local view of the computation, and would be unable to cope with limitations in the presence of in-place array updates, or fuse if an array operation is used multiple times. The Glasgow Haskell Compiler itself also bases its list fusion on rewrite rules and cross-module inlining [JTH01].

The Repa [Kel+10] approach to fusion is based on a delayed representation of arrays at run-time, which models an array as a function from index to value. With this representation, fusion happens automatically through function composition, although this can cause duplication of work in many cases. To counteract this, Repa lets the user *force* an array, by which it is converted from the delayed representation to a traditional sequence of values. The pull arrays of Obsidian [CSS12] use a similar mechanism.

Accelerate [McD+13] uses an elaboration of the delayed arrays representation from Repa, and in particular manages to avoid duplicating work. All array operations have a uniform representation as constructors for delayed arrays, on which fusion is performed by tree contraction. Accelerate supports multiple arrays as input to the same array operation (using a `zipWith` construct). Although arrays are usually used at least twice (once for getting the size, once for the data), it does not seem that Accelerate handles the difficult case where the output of an array operation is used as input to two other array operations.

NESL has been extended with a GPU backend [BR12], for which the authors note that fusion is critical to the performance of the flattened program. The NESL approach is to use a form of copy-propagation on the intermediary code, and lift the resulting functions to work on entire arrays. Their approach only works for what we would term **`map`**-**`map`** fusion, however.

The *with*-loops of SaC can fulfill the same role as **`redomap`** [GHS06], although the fusion algorithm in which they are used is very different: in SaC, producer-consumer and horizontal fusion is combined in a general framework of *with-loop fusion*. The *with*-loops can not, however, fulfill the role taken by **`stream_seq`** or **`stream_par`** in permitting efficient sequentialisation.

Chapter 8

Moderate Flattening and Kernel Extraction

This chapter presents a transformation that aims to enhance the degree of statically-exploitable parallelism by reorganizing imperfectly nested parallelism into perfect SOAC nests. The outer levels of the resulting nestings correspond to **`map`** operators, which are trivial to map to GPUs, and the innermost one is an arbitrary SOAC or sequential code. In essence, the transformation seeks to rewrite the program to express patterns of parallelism that have a known efficient implementation to GPUs. The simplest such pattern is a perfect nest of **`map`**s, where the innermost function contains arbitrary sequential code. This pattern maps directly to a single GPU kernel, but there are other interesting patterns that correspond to, for example, segmented scans and reductions.

In a purely functional setting, Blelloch's transformation [Ble90] flattens all available parallelism, while asymptotically preserving the depth and work of the original nested-parallel program. In our setting, this corresponds to interchanging all sequential loops outwards, and forming **`map`**-nests that contain only simple scalar code. While asymptotically efficient, the approach is arguably inefficient in practice [BR12], for example because it does not account for locality of reference, and pays a potentially large cost in memory usage to extract parallelism that may not be necessary. For example, a fully flattened implementation of matrix multiplication requires $O(n^3)$ storage, where the usual implementation uses only $O(n^2)$ [Spo+08].

Our algorithm, presented below, builds on **`map`**-**`loop`** interchange and **`map`** distribution, exploiting the property that it is always safe to interchange a parallel loop inwards, or to distribute a parallel loop [KA02]. Our algorithm attempts to exploit some of the efficient *top-level parallelism*. Two important limitations are:

1. We do not exploit parallelism inside **`if`** branches, as this generally requires expensive **`filter`** operations.

2. We terminate distribution when it would introduce irregular arrays, as these obscure access patterns and prevent further spatial- and temporal-locality optimizations.

Our algorithm is less general than full flattening, but generates more analysable code in the common case, for programs that do not require the generality of flattening. Also, our presented algorithm is not *cost-preserving*, in the notion of Blelloch [BG96]. Should we assign a NESL-style cost model to Futhark, the result of our moderate flattening algorithm would often have a different asymptotic cost (less parallelism) than the original program.

On the upside, moderate flattening enables the optimisations covered in Chapter 9. Multi-versioned code, discussed in Section 8.3, could be used to apply full flattening as a fallback for those cases where moderate flattening is incapable of extracting sufficient parallelism. However, this technique is not yet implemented.

8.1 Example of Moderate Flattening

Figures 56a and 56b demonstrate the application of our algorithm on a contrived but illustrative example that demonstrates many of the flattening rules exploited in the generation of efficient code for the various benchmark programs. The original program consists of an outer **map** that encloses (i) another **map** operator implemented as a sequence of **map**s, **reduce**s, and **scan**s and (ii) a loop containing a **map** whose implementation is given by a **reduce** and some scalar computation. As written, only one level of parallelism (for example, the outermost) can be statically mapped on GPGPU hardware. Our algorithm distributes the outer **map** across the enclosed **map** and **loop** bindings, performs a **map**-**loop** interchange, and continues distribution. The result consists of four perfect nests: a **map**–**map** and **map**–**map**–**map** nest at the outer level, and a **map**–**map**–**reduce** (segmented reduction) and **map**–**map** nest contained inside the loop. In the first **map**–**map** nest, the **scan** and **reduce** are sequentialized because further distribution would generate an irregular array, as the size `p` of `cs` is variant to the second **map**.

8.2 Rules for Moderate Flattening

Figure 57 lists the rules that form the basis of the flattening algorithm. We shall use Σ to denote *map nest contexts*, which are sequences of *map contexts*, written $\mathbf{M}\ \overline{x}\ \overline{y}$, where $\overline{x}$ denotes the bound variables of the map operator over the arrays held in $\overline{y}$. The flattening rules, which take the form $\Sigma \vdash e \Rightarrow e'$, specify how a source expression e may be translated into an equivalent target expression e' in the given map nest context Σ. Several rules may be applied in each situation. The particular algorithm used by Futhark bases its decisions on crude heuristics related to the structure of the map

```
let (asss, bss) =
 map
  (λps: ([m][m]i32,[m]i32) →
   let ass =
    map
     (λp: [m]i32 →
      let cs =
        scan (+) 0 (iota p)
      let r = reduce (+) 0 cs
      let as = map (+r) ps
      in as) ps
   let bs =
    loop (ws=ps) for i < n do
     let ws' =
      map
       (λas w: i32 →
        let d = reduce (+) 0 as
        let e = d + w
        let w' = 2 * e
        in w') ass ws
     in ws'
   in (ass, bs))
  pss
```

(a) Program before distribution.

```
let rss =
 map
  (λps: [m]i32 →
    map
     (λp: i32 →
      let cs = scan (+) 0 (iota p)
      let r = reduce (+) 0 cs
      in r) ps) pss
let asss =
 map (λps rs: [m]i32 →
       map (λr →
             map (+r) ps)
           rs)
     pss rss
let bss =
 loop (wss=pss) for i < n do
  let dss =
   map (λass: [m]i32 →
         map (λas: i32 →
               reduce (+) 0 as)
             ass)
       asss
  in map (λws, ds: [m]i32 →
           map (λw d: i32 →
                 let e = d + w
                 let w' = 2 * e
                 in w')
               ws ds)
         wss dss
```

(b) Program after distribution.

Figure 56: Extracting kernels from a complicated nesting. We assume `pss` : $[m][m]$`i32`. Exploitable (outermost or perfectly nested) levels of parallelism are highlighted in blue.

nest context and the inner expression. Presently, nested **`stream_par`**s are sequentialised, while nested **`map`**s, **`scan`**s, and **`reduce`**s are parallelised. These rules were mainly chosen to exercise the code generator, but sequentialising **`stream_par`** is the right thing to do for most of the data sets we use in Section 10.

For transforming the program, the flattening algorithm is applied (in the empty map nest context) on each map nest in the program. Rule G1 (together with rule G3) allows for manifestation of the map nest context Σ over e. Whereas rule G1 can be applied for any e, the algorithm makes use of this rule only when no other rules apply. Given a map nest context Σ and an instance of a `map` SOAC, rule G2 captures the `map` SOAC in the map nest context. This rule is the only rule that extends the map nest context.

Rule G4 allows for map fission (**`map`** $(f \circ g) \Rightarrow$ **`map`** $f \circ$ **`map`** g), in the sense that the map nest context can be materialized first over e_1 and then over e_2 with appropriate additional context to allow for access to the now array-materialized values that were previously referenced through the let-bound variables $\overline{a_0}$. The rule can be

Basic Flattening Rules $\boxed{\Sigma \vdash e \Rightarrow e}$

$$\frac{\Sigma \vdash \mathbf{map}\ (\lambda\ \overline{x}\ \rightarrow\ e)\ \overline{y} \Rightarrow e'}{\Sigma, \mathbf{M}\ \overline{x}\ \overline{y} \vdash e \Rightarrow e'} \tag{G1}$$

$$\frac{\Sigma, \mathbf{M}\ \overline{x}\ \overline{y} \vdash e \Rightarrow e'}{\Sigma \vdash \mathbf{map}\ (\lambda\ \overline{x}\ \rightarrow\ e)\ \overline{y} \Rightarrow e'} \tag{G2}$$

$$\frac{}{\emptyset \vdash e \Rightarrow e} \tag{G3}$$

$$\frac{\begin{array}{c}\Sigma = \mathbf{M}\ \overline{x_p}\ \overline{y_p}, \ldots, \mathbf{M}\ \overline{x_1}\ \overline{y_1} \\ \Sigma' = \mathbf{M}\ (\overline{x_p}, \overline{a_{p-1}})\ (\overline{y_p}, \overline{a_p}), \ldots, \mathbf{M}\ (\overline{x_1}, \overline{a_0})\ (\overline{y_1}, \overline{a_1}) \\ \overline{a_p}, \ldots, \overline{a_1} \text{ fresh names} \\ \text{size of each array in } \overline{a_0} \text{ invariant to } \Sigma \\ \Sigma \vdash e_1 \Rightarrow e_1' \quad \Sigma' \vdash e_2 \Rightarrow e_2'\end{array}}{\Sigma \vdash \mathbf{let}\ \overline{a_0} = e_1\ \mathbf{in}\ e_2 \Rightarrow \mathbf{let}\ \overline{a_p} = e_1'\ \mathbf{in}\ e_2'} \tag{G4}$$

$$\frac{\begin{array}{c}g = \mathbf{reduce}\ (\lambda\ \overline{y}^{2*p}\ \rightarrow\ e)\ \overline{n}^p \\ \Sigma \vdash \mathbf{map}\ (g)\ (\mathbf{transpose}\ z_0) \ldots (\mathbf{transpose}\ z_{p-1}) \Rightarrow e' \\ f = \mathbf{map}\ (\lambda\ \overline{y}^{2*p}\ \rightarrow e)\end{array}}{\Sigma \vdash \mathbf{reduce}\ (f)\ (\overline{\mathbf{replicate}\ k\ n}^p)\ \overline{z}^p \Rightarrow e'} \tag{G5}$$

$$\frac{\Sigma \vdash \mathbf{rearrange}\ (0, 1 + k_0, \ldots, 1 + k_{n-1})\ y \Rightarrow e}{\Sigma, \mathbf{M}\ x\ y \vdash \mathbf{rearrange}\ \overline{k}^{(n)}\ x \Rightarrow e} \tag{G6}$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma, \mathbf{M}\ (\overline{x}, \overline{y})\ (\overline{xs}, \overline{ys}) \qquad (\{n\} \cup \overline{q}) \cap (\overline{x}, \overline{y}) = \emptyset \\ m = \text{outer size of each of } \overline{xs} \text{ and } \overline{ys} \\ \text{f contains exploitable (regular) inner parallelism} \\ \Sigma \vdash \mathbf{loop}\ \ = (\ )\mathbf{for}\ \overline{zs'} = \overline{\mathbf{replicate}\ m\ z_i}, \overline{ys'} = \overline{ys} < \mathbf{do} \\ \qquad \mathbf{for}\ i < n\ \mathbf{do}\ \mathbf{map}(f\ i\ \overline{q})\ \overline{xs}\ \overline{ys}\ \overline{ys'}\ \overline{zs'} \Rightarrow e\end{array}}{\Sigma' \vdash\ \mathbf{loop}\ \ = (\ )\mathbf{for}\ \overline{z'} = \overline{z},\ \overline{y'} = \overline{y} <\ \ \mathbf{do} \\ \mathbf{for}\ i < n\ \mathbf{do}\ f\ i\ \overline{q}\ \overline{x}\ \overline{y}\ \overline{y'}\ \overline{z} \Rightarrow e} \tag{G7}$$

Figure 57: The flattening rules that form the basis of the moderate flattening algorithm.

applied only if the intermediate arrays formed by the transformation are ensured to be regular, which is enforced by a side condition in the rule. To avoid unnecessary and excessive flattening on scalar computations, the **let**-expressions are rearranged using a combination of **let**-floating [PPS96] and tupling for grouping together scalar code in a single **let**-construct. In essence, inner SOACs are natural splitting points for fission. For example,

```
let b = x+1
let a = b+2
in replicate n a
```

is conceptually split as

```
let a = (let b = x+1 in b+2)
in replicate n a
```

Rule G5 allows for **reduce**-**map** interchange where it is assumed that the source neutral reduction element is a replicated value. The original pattern appears in k-means (see Figure 8) as a reduction with an array operator, which is inefficient on a GPU if executed as such. The interchange results in a segmented-reduce operator (applied on equally-sized segments), at the expense of transposing the input array(s). This rule demonstrates a transformation of both schedule and (if the transposition is manifested) data of the program being optimized.

Rule G6 allows for distributing a **rearrange** construct by rearranging the outer array (input to the map nest) with an expanded permutation. The semantics of **rearrange** `p a` is that it returns `a` with its dimensions reordered by a statically-given permutation **p**. For instance, the expression **rearrange** `(2,1,0) a` reverses the dimensions of the three-dimensional array `a`. For convenience, we use `transpose a` as syntactic sugar for **rearrange** `(1,0,...) a`, which swaps the two outermost dimensions. Similar rules can be added to handle other expressions that have a particularly efficient formulation when distributed on their own, such as concatenation, but we elide these for brevity.

Finally, rule G7 implements a **map**-**loop** interchange. The simple intuition is that

map (λ`x` $\rightarrow$ **loop** `(x'=x)` **for** `i < n` **do** `(f x'))` `xs`

is equivalent to

loop `(xs'=xs)` **for** `i < n` **do** (**map** `f xs'`)

because they both produce `[f`^{n}`(xs[0]), ... ,f`^{n}`(xs[m-1])]`. The rule is sufficiently general to deal with all variations of variant and invariant variables in the **loop** body. The side condition in the rule ensures that $z \subseteq q$ are free variables and thus invariant to Σ. The rule is applied only if the body of the loop contains inner parallelism, such as maps, otherwise its application is not beneficial (as an example, it would change the Mandelbrot benchmark from Section 10.1.4 to have a memory-

rather than a compute-bound behavior). However, rule G7 is essential for efficient execution of the LocVolCalib benchmark (Section 10.1.3), because a loop separates the outer map from four inner maps.

We conclude by remarking that some of the choices made in the flattening rewrite rules about how much parallelism to exploit and how much to sequentialize efficiently are arbitrary, because there is no size that fits all. For example, we currently sequentialize a **`stream_par`** if it is inside a **`map`** nest, but the algorithm can easily be made more aggressive. To see the problem, consider an expression that computes the sum of rows of a matrix:

```
map (λxs → reduce (+) 0 xs) xss
```

Assume that `xss` is of shape $n \times m$. If n is sufficiently large (tens of thousands on a GPU), then it is most efficient to turn the **`reduce`** into a sequential loop. If n is very small, then perhaps it is best to sequentialise the **`map`**, and keep the **`reduce`** parallel. Or perhaps we must exploit the parallelism of both the **`map`** and the **`reduce`** to fully saturate the parallel capacity of the target hardware. Since n and m are not known at compile-time, we cannot generate code that is optimal for all cases.

A more general solution would be to generate all three possible code versions, and to discriminate between them at runtime based on dynamic predicates that test whether the exploited parallelism is enough to fully utilize hardware. This idea is discussed further below.

8.3 Multi-Versioned Code

The flattening rules of the preceding section are able to directly decompose **`map`**-nestings via loop distribution. However, they are not in general able to exploit parallelism nested within the functional argument to a non-**`map`** SOAC. Some rules, for example G5 that performs **`reduce`**-**`map`** interchange, expose **`map`**s which can then be further handled by the moderate flattening algorithm. This section presents further rules that decompose SOACs into **`map`**s and other SOACs. Furthermore, we also introduce a rule for *multi-versioned code*, by which a single SOAC may be flattened in two different ways, with either of the two alternatives picked at runtime via a branch. While the Futhark compiler contains a prototype implementation of multi-versioned code, it is not yet used by default. Hence, the effectiveness of the ideas in this section are not shown empirically on benchmarks.

Multi-versioned code does not change the overall framework of moderate flattening. Rather, Figure 58 simply provides additional rules that are more aggressive in decomposing SOACs into constituent parts. The most crucial rule is E1, which states that if some expression e can be flattened in two distinct ways e'_1 and e'_2, then e can be flattened to an **`if`** expression that uses some runtime *predicate* to determine which of e'_1 or e'_2 to apply. We do not specify the form of the predicate in detail here. One typical form is to somehow quantify the amount of parallelism exploited by e'_1 versus e'_2, and picking the version that exploits the *least* amount of parallelism that still

More Flattening Rules $\boxed{\Sigma \vdash e \Rightarrow e}$

$$\frac{\begin{array}{c}\Sigma \vdash e \Rightarrow e'_1 \\ \Sigma \vdash e \Rightarrow e'_2 \\ e'_1 \neq e'_2 \\ v \text{ is fresh}\end{array}}{\Sigma \vdash e \Rightarrow \texttt{let } v = \mathit{predicate} \texttt{ in if } v \texttt{ then } e'_1 \texttt{ else } e'_2} \quad \text{(E1)}$$

$$\frac{\begin{array}{c}\overline{a}^{(n)}\ \overline{b}^{(m)}\ \overline{c}^{(n)} \text{ are fresh} \\ f' = f\ \overline{v}^{(n)} \\ op_1 \equiv \texttt{map}\ f'\ \overline{x} \quad op_2 \equiv \texttt{reduce}\ \oplus\ (\overline{v}^{(n)})\ \overline{a}^{(n)} \\ \Sigma \vdash \texttt{let}\ \overline{a}^{(n)}\ \overline{b}^{(m)} = op_1 \texttt{ in let } \overline{c}^{(n)} = op_2 \texttt{ in } (\overline{c}^{(n)}, \overline{b}^{(m)}) \Rightarrow e'\end{array}}{\Sigma \vdash \texttt{redomap}\ \oplus\ f\ (\overline{v}^{(n)})\ \overline{x} \Rightarrow e'} \quad \text{(E2)}$$

$$\frac{\begin{array}{c}\overline{a}^{(n)}\ \overline{b}^{(m)}\ \overline{c}^{(n)} \text{ are fresh} \\ op_1 \equiv \texttt{map}\ (\text{unchunk}(n, m, f))\ \overline{x} \quad op_2 \equiv \texttt{reduce}\ \oplus\ (\overline{v}^{(n)})\ \overline{a}^{(n)} \\ \Sigma \vdash \texttt{let}\ \overline{a}^{(n)}\ \overline{b}^{(m)} = op_1 \texttt{ in let } \overline{c}^{(n)} = op_2 \texttt{ in } (\overline{c}^{(n)}, \overline{b}^{(m)}) \Rightarrow e'\end{array}}{\Sigma \vdash \texttt{stream\_par}\ \oplus\ f\ \overline{x} \Rightarrow e'} \quad \text{(E3)}$$

$$\frac{\begin{array}{c}w \text{ is size of any } x \\ \Sigma \vdash \texttt{let } c = w \texttt{ in } \overline{\texttt{let } v_i = x_i}^{(k)} \texttt{ in } e_f \Rightarrow e'\end{array}}{\Sigma \vdash \texttt{stream\_seq}\ (\lambda c\ \overline{v}^{(k)}\ \rightarrow e_f)\ (\overline{v}^{(n)})\ \overline{x}^{(k)} \Rightarrow e'} \quad \text{(E4)}$$

Figure 58: Flattening rules that are more aggressive in exploiting inner parallelism. The definition of unchunk can be found on Figure 59.

$$\begin{aligned}\text{unchunk}(n, m, \lambda c\ \overline{v}^{(k)} :\ (\overline{\tau}^{(n)}, \overline{[c]\tau}^{(m)})\ \rightarrow e) = \lambda \overline{v'}^{(k)} : (\overline{\tau}^{(n)}, \overline{\tau}^{(m)}) \rightarrow\ & \\ & \texttt{let } c\ =\ 1 \\ & \overline{\texttt{let } v_i\ =\ [v'_i]}^{(k)} \\ & \texttt{let } \overline{a}^{(n)}\ \overline{b}^{(m)}\ =\ e \\ & \overline{\texttt{let } b'_i\ =\ b_i[0]}^{(m)} \\ & \texttt{in } \overline{a}^{(n)}\ \overline{b'}^{(m)}\end{aligned}$$

Figure 59: Turning a chunk function into a non-chunked function by setting the chunk size to unit, and turning the resulting unit-size map-out results into single values.

exceeds some runtime-given constant that characterises the hardware the program is executing on. The intuition is that we want to fully saturate the hardware, but parallelism in excess of that is a waste. Determining the right form of the predicates, and the constants they may employ, is in practice a form of *auto-tuning*.

Rule E2 describes how a **redomap** can be taken apart into a **map** and a **reduce**, which is then (potentially) further flattened. The **map** uses a modified function f', in which we fix the first n parameters of the original function f (corresponding to the accumulator) to the neutral value. Since **redomap** returns both reduced and mapped results, we are careful to only pass the former to the **reduce**. An almost identical rule can be defined for **scanomap**.

Rule E3 is similar to E2, but applies to **stream_par**, and recovers all available parallelism by splitting into a **map** and a **reduce**. The main difference compared to E2 is that computing the function for **map** is somewhat more complicated, as shown on Figure 59.

Rule E4 is used to exploit any parallelism inside of a **stream_seq**. This is done by fixing the chunk size c to the full size w of the input arrays, and then inserting the function body.

8.4 Kernel Extraction

The moderate flattening algorithm, as presented in this chapter, does not directly transform SOACs into flat-parallel kernels suitable for GPU execution. Instead, it merely rearranges the SOACs into simpler forms that are passed to a simple pattern matcher, which recognises forms of *top-level* parallelism, and transforms it to an explicitly flat kernel representation of GPU kernels. In this context, top-level parallelism is a SOAC that is not embedded inside the function of another SOAC. We will not describe this operation, or the full kernel representation used by the Futhark compiler, in detail, as it pertains to straightforward but tedious book-keeping, but some details are worth mentioning.

In the implementation, there is no distinction between moderate flattening and kernel extraction: it is interleaved in the same pass. Conceptually, when rule G1 first fires, it will sequentialise the expression e, and then manifest the *entire* map context Σ at once, in the form of a flat parallel kernel.

Not all kernels are in the form of a **map** nest containing a sequential function. Other patterns that may give rise to a GPU kernel are:

- **redomap** or **scanomap** can be transformed into two GPU kernels via the techniques used in Section 4.5.
- **stream_par** can be transformed into two GPU kernels similarly to a **redomap**. Any parallelism inside the functional arguments is sequentialised.

- **`redomap`** or **`scanomap`** perfectly nested within one or more **`map`**s can be transformed into kernels implementing segmented reduction or prefix scan. The number of kernels depends on the exact implementation strategy, which is outside the scope of this thesis. The implementation strategy for segmented reductions used by the Futhark compiler is presented in [LH17].
- **`scatter`** can be transformed directly into a single GPU kernel similar to **`map`**.

The bodies of the kernels produced by kernel extraction remain Futhark code, and kernel extraction maintains the original structure of any sequential code. Importantly, even though nested SOACs may not have had their potential for parallelism exploited, they remain high-level descriptions of loops, which can be used for further optimisation. We shall see examples in Chapter 9.

Later in the compiler, the high-level kernel representation is transformed into a low-level imperative IR, and from there to actual OpenCL code. However, these topics are outside the scope of this thesis.

8.5 Related Work

Futhark builds on previous work in type systems and parallel compilation techniques. Recent work [Ste+15] shows that stochastic combinations of rewrite rules open the door to autotuning. Work has been done on designing purely functional representations for OpenCL kernels [SRD17], which could in principle be targeted by our moderate flattening algorithm. The Futhark compiler presently uses a similar (but simpler) representation, the full details of which are outside the scope of this thesis.

Halide [Rag+13] uses a stochastic approach for finding optimal schedules for fusing stencils by a combination of tiling, sliding window and work replication. This is complementary to Futhark, which does not optimise stencils, nor uses autotuning techniques, but could benefit from both. In comparison, Futhark supports arbitrary nested parallelism and flattening transformation, together with streaming SOACs that generically encode strength-reduction invariants.

Delite uses rewrite rules to optimize locality in NUMA settings [Bro+16] and proposes techniques [Lee+14] for handling simple cases of nested parallelism on GPUs by mapping inner parallelism to CUDA block and warp level. We use transposition to handle coalescing (see Chapter 9), and, to our knowledge, no other compiler matches our AST-structural approach to kernel extraction, except for those that employ full flattening, such as NESL [Ble+94; BR12], which often introduces inefficiencies and does not support in-place updates. Data-only flattening [Ber+13] shows how to convert from nested to flat representation of *data*, without affecting program structure. This would be a required step in extending Futhark to exploit irregular parallelism. In comparison, Futhark flattens some of the top-level parallelism *control*, while preserving the inner structure and opportunities for locality-of-reference optimizations.

Imperative GPU compilation techniques rely on low-level index analysis ranging from pattern-matching heuristics [Yan+10; Dub+12] to general modeling of affine transformations by polyhedral analysis [Ver+13; Pou+11]. Since such analyses are often hindered by the expressivity of the language, solutions rely on user annotations to improve accuracy. For example, OpenMP annotations can be used to enable transformations of otherwise unanalyzable patterns [CSS15], while PENCIL [Bag+15] provides a restricted C99-like low-level language that allows the (expert) user to express the parallelism of loops and provide additional information about memory access patterns and dependencies.

The moderate flattening transformation resembles the tree-of-bands construction [Ver+13] in that it semantically builds on interchange and distribution, but we use rules that directly exploit properties of the language. For example, imperative approaches would implement a reduction with a vectorized operator via a histogram-like computation [RKC16], which is efficient only when the histogram size is small. In comparison, rule G5 in Figure 57 transforms a reduction with a vectorized operator to a (regular) segmented reduction, which always has an efficient implementation.

Chapter 9

Optimising for Locality of Reference

While GPUs possess impressively fast memory, the ratio of computation to memory speed is even more lopsided than on a CPU. As a consequence, accessing memory efficiently is of critical importance to obtaining high performance. Furthermore, while CPUs have multiple layers of automatic caching to help mitigate the effect of the memory wall, GPUs typically have only a single very small level-1 cache. This chapter presents how two well-known locality-of-reference optimisations can be applied to kernels produced by moderate flattening. The optimisations improve the memory access patterns of Futhark programs by optimising both spatial and temporal locality of reference.

Section 9.1 shows how arrays traversed in a kernel can have their representation in memory modified to ensure that the traversal is efficient. This form of optimisation for spatial locality of reference, which changes not just the *code*, but also the *data*, is not usually used by compilers. Section 9.2 shows a technique for temporal locality of reference, via loop tiling, that uses the local memory of the GPU as a cache to minimise the amount of traffic to global memory. While loop tiling is an established optimisation technique, our approach can handle even indirect accesses. These are not supported by conventional techniques for loop tiling, including polyhedral approaches such as the ones discussed in [CSS15].

To express the optimisations we have to extend the core Futhark IR, yet again, with a few more operations. Their types are shown on Figure 60, and their semantics below.

`manifest` $(\overline{c}^{(n)})$ x is used to force a particular representation of an n-dimensional array in memory. The permutation $(c_1, \ldots, c_n)$ indicates the order in which dimensions should occur. Thus, for the two-dimensional case,

```
manifest (0,1) xss
```

corresponds to a row-major (the default) copy of the array `xss`, while

```
manifest (1,0) xss
```

is a column-major copy. Note that the type of the result of **manifest** is the same as of the input array. The only change is in how the array is located in memory, and thus the memory address computed when indexing an element at a given index. As with **rearrange**, the permutation must be a compile-time constant. The **manifest** construct differs from **rearrange** in that **manifest** creates an actual array in memory, while **rearrange** is merely a symbolic index space transformation.

kernel $(\overline{d}^{(n)})$ $(\lambda\overline{v}^{(2n)} : \overline{\tau}^{(m)} \to e)$ models an n-dimensional GPU kernel, loosely corresponding to a flattened form of n **map**-nests, where nest number i (counted from outside-in) has size d_i. Semantically, the kernel function is executed for every thread, and passed n *global thread indices*, one for every dimension, and n *local thread indices*, again one for every dimension (see Section 4.1)—this is why the lambda accepts $2n$ parameters. In many cases we will not be making use of the local thread indices, in which case we will simply put an underscore for the parameter names. The group size is not explicitly stated, but may be implicitly bounded through the use of **local** (see below).

If the kernel function returns m values, then the **kernel** construct as a whole returns m arrays, each of of shape $[d_1]\cdots[d_n]$. This corresponds to each thread returning a single value.

local $\overline{c}^{(n)}$ x may only occur within **kernel**s, and produces an array of shape $[c_1]\cdots[c_n]$ where every thread contributes a value x. Operationally, $\overline{c}^{(n)}$ corresponds to the group size of the kernel (and any kernel containing a **local** is thus implicitly forced to have that group size), and the resulting array is located in local memory. The subtleties of this construct are detailed further in Section 9.2.

op		TySch(*op*)
manifest $(c_1,\ldots,c_n)$	:	$\forall\overline{d}^{(n)}\alpha.[d_1]\cdots[d_n]\alpha \to [d_1]\cdots[d_n]\alpha$
kernel	:	$\forall\overline{\alpha}^{(m)}.\overline{(d_i : \texttt{i32})}^{(n)}$
		$\to (\overline{\texttt{i32}}^{(2n)} \to (\overline{\alpha}^{(m)}))$
		$\to (\overline{[d_1]\cdots[d_n]\alpha}^{(m)})$
local c	:	$\forall\alpha.\overline{(d_i : \texttt{i32})}^{(n)} \to \alpha \to [d_1]\cdots[d_n]\alpha$

Figure 60: The types of the language constructs used to express kernel-level locality of reference optimisations.

9.1 Transposing for Coalesced Memory Access

Ensuring coalesced accesses to global memory is critical for GPU performance. Several of the benchmarks discussed in Chapter 10, such as FinPar's LocVolCalib, Accelerate's n-body, and Rodinia's CFD, k-means, Myocyte, and LavaMD, exhibit kernels in which one or several innermost dimensions of arrays are processed sequentially inside the kernel. In the context of the moderate flattening algorithm, this typically corresponds to the case where rule G1 has been applied with e being a SOAC. For this discussion, we assume that the nested SOAC is transformed to a **`stream_seq`**; removing parallelism, but retaining access pattern information.

A naive translation of a nested **`stream_seq`** would lead to consecutive threads accessing global memory with a stride equal to the size of the inner (non-parallel) array dimensions, which may generate one-order-of-magnitude slowdowns on a GPU. The Futhark compiler solves this by, intuitively, transposing the non-parallel dimensions of the array outermost, and the same for the result and all intermediate arrays created inside the kernel. For this thesis, we only discuss the first case, of arrays that are sequentially iterated inside a kernel, as the others are not a question of transforming an AST, but simply a question of how we allocate arrays in the first place. Our approach is guaranteed to resolve coalescing if the sequential-dimension array indices are invariant to the parallel array dimensions. An array index is invariant to a dimension if each thread along that dimension will have the same value for that index. Or put another way, an array index is invariant to a dimension if the value of the index does not depend on the global or local thread index alongside that dimension. For example, consider the following expression:

```
kernel (n) (λi _ → stream_seq f (0) xss[i])
```

Assuming that `f` performs a sequential in-order traversal of its input, the expression is optimized by changing the representation of `xss` to be column major (the default is row major), via transposition in memory, as follows:

```
let xss' = manifest (1,0) xss
in kernel (n) (λi _ → stream_seq f (0) xss'[i])
```

The type of `xss'` is the same as that of `xss`. The difference between `xss` and `xss'` can be seen by computing the flat index corresponding to an index expression. Suppose `xss` has shape $[n][m]$, i is the parallel (thread) index, j is the counter of a sequential loop, and $\text{flat}(x)$ is a function that returns a one-dimensional view of an array x; then

$$\texttt{xss}[i][j] = \text{flat}(xss)[i \cdot m + j]$$

versus

$$\texttt{xss'}[i][j] = \text{flat}(xss')[j \cdot m + i].$$

It is clear that the latter gives rise to coalesced memory accesses, while the former does not (assuming non-pathological values of m).

Concretely, the compiler inspects index expressions $x[\overline{y}^{(n)}]$ inside of kernels, where x is a rank m array that is created prior to the kernel. The inspection may yield a permutation $\overline{c}^{(m)}$ that is used to construct an array

$$x' = \textbf{manifest}\ (\overline{c}^{(m)})\ x$$

after which x is replaced by x' in the original index expression. Two patterns are recognised by the inspection:

Complete Index

The indices $\overline{y}^{(n)}$ contain at least some of the the global thread indices, and the innermost global thread index can be moved to the last position in the indices via the permutation $\overline{c}^{(m)}$. We then copy the array with permutation $\overline{c}^{(m)}$. An example is a kernel

```
kernel (n,m) (λi j _ _ → xss[j][i] + 2)
```

which is transformed into

```
let xss' = manifest (1,0) xs
in kernel (n,m) (λi j _ _ → xss'[j][i] + 2)
```

For this example, we could also have transposed the kernel dimensions themselves, followed by transposing the result:

```
let r = kernel (m,n) (λj i _ _ → xss'[j][i] + 2)
in rearrange (1,0) r
```

This saves on memory traffic, since **rearrange** is a non-manifest index space transformation. However, this transformation affects the memory layout of the result of the kernel, which may result in non-coalesced accesses in later expressions. As a result, we prefer using **manifest**, which has purely local effects.

Note that the pattern also applies to the case where the found permutation is the identity permutation. A useful optimisation is to remove those **manifest**s where it can be determined that the array is already in the desired representation, either because of a **rearrange** that was already present in the source program, or because the optimal representation is row-major, which is generally the default for Futhark expressions.

Incomplete Index

The indices $\overline{y}^{(n)}$ do not fully index the array ($n < m$), and at least one of the indices is variant to the thread indices. An array index y_i is variant to a thread index if there is a (conservatively estimated) data dependency from one of the thread indices passed to the kernel function to y_i. In such cases, we use **`manifest`** with the permutation $(n, \ldots, m-1, 0, \ldots, n-1)$, corresponding to interchanging the sequentially traversed dimensions outermost.

The (potentially wrong) assumption is that the dimensions that are not indexed will be traversed sequentially. It remains future work to perform a more detailed analysis of how the resulting array slice is traversed.

The requirement that at least one of $\overline{y}^{(n)}$ must be thread-variant is to avoid the pattern triggering on cases where an array is being indexed identically by all threads, such as `xss[0]`. Such indexes do not give rise to non-coalesced memory access.

Note that the transformation is on index expressions, not on arrays. The same array `xss` may be indexed in different ways within the same kernel, and each distinct index expression may give rise to a unique new **`manifest`** array. However, if the same array is iterated in the same way in multiple places within the kernel, this will still give rise only to a single array.

Our approach is not necessarily optimal—there are cases where several distinct manifestations (as given by the rules above) all solve the specific cases of uncoalesced access, but so might a combined *single* manifestation, with an appropriately selected permutation. This would require the algorithm to take a more "global" view, and has not yet been done.

9.1.1 Viability of the Optimisation

This optimisation requires both additional work at runtime, as well as extra memory space to perform transpositions before the kernel is invoked. The Futhark compiler presently always applies the optimisation whenever it detects an array indexing of the appropriate form. A relevant question to ask is whether there are cases where this is not an optimisation, but rather a *pessimisation*. The short answer: in some cases, yes, but judging by our benchmarks (Chapter 10), these occur rarely in practice.

First we remark that transpositions are quite fast on GPUs; running at close to the speed of a memory copy. Therefore, even in pathological cases where one of the dimensions is unit, or the inner sequential loop is so short that the uncoalesced accesses have no great impact, the cost of the transpositions is not catastrophic.

A greater concern is having to allocate extra memory to store the transposed copies. In the worst case, this may require temporarily migrating the original arrays to CPU memory in order to fit the new copies. The CPU-GPU bandwidth is very low (current PCIe 4.0 x16 runs at 31.51GiB/s in the best case), so this could lead to

significant slowdown. In the future, we intend to investigate how to "back-propagate" the **manifest** operations to the point at where the array is originally created, and store it in the desired representation in the first place.

9.1.2 Indirect Accesses

The coalescing transformation applies even in cases where indirect accesses are involved, as it depends only on variance information. For example, consider this variation on the earlier example, where an array `js` of indices is used to index `xss`.

```
let (js: [n]i32) = ...
let (xss: [m][k]i32) = ...
in kernel (m) (λi _ →
  loop acc = 0 for p < n do
    let j = js[p]
    let x = xss[i,j]
    in acc + x)
```

To the coalescing optimisation, this kernel contains two array indices: `js[p]` and `xss[i,j]`. The former is not touched, since it does not involve any thread indices, and so does not meet either of the rules. However, the indexing expression `xss[i,j]` *does* match the rule for complete indexes, since the global thread index `i` can be moved to the innermost position via the permutation `(1,0)`. Thus, the kernel is transformed to the following, where all accesses are coalesced.

```
let (js: [n]i32) = ...
let (xss: [m][k]i32) = ...
let (xss': [m][k]i32) = manifest (1,0) xss
in kernel (m) (λi _ →
  loop acc = 0 for p < n do
    let j = js[p]
    let x = xss'[i][j]
    in acc + x)
```

Note that the `p` index in `js[p]` is invariant to the kernel, and as such leads to effective caching (each thread in a warp accesses the same `js[p]` in the same SIMD instruction).

9.2 Automatic Loop Tiling

The Futhark compiler performs simple block *tiling* in cases where several threads are traversing the same array. The tiling algorithm transforms traversal to be over *chunks*

of the input, where each chunk is collectively copied to local memory before it is traversed. The result is that the number of accesses to global memory is reduced.

The algorithm is driven by recognizing arrays that are inputs to **stream_seq** constructs *and* are invariant to one or two of the kernel's innermost dimensions. The tiling algorithm proceeds by traversing the body of a **kernel** and looking for an appropriate **stream_seq** instance, which is then transformed using either *one-dimensional* or *two-dimensional* tiling, as detailed below. In both cases, the **stream_seq** is transformed into two nested **stream_seq**s, with the outer one iterating across chunks, and the inner iterating across a single chunk. We tile at most one **stream_seq** for a kernel, as each instance of tiling may impose constraints on the group size of the GPU kernel, and it is not clear whether multiple instances of tiling would lead to conflicting constraints.

In the following, we say that an array is invariant to a parallel dimension if there is no data dependency between the array and the thread index corresponding to that dimension. To simplify the explanation, we are ignoring a significant amount of tedious complexity related to computing GPU group sizes and handling cases where the tiled arrays have sizes that are not divisible with the optimal tile size.

9.2.1 One-Dimensional Tiling

One-dimensional tiling is performed when the input arrays of a **stream_seq** are invariant to the same dimension. We exploit a *stripmining property* of **stream_seq**, by which any **stream_seq** can be transformed into two nested **stream_seq**s.

For example, the kernel

```
kernel (n) (λi li → stream_seq (f i) (v) ps)
```

can have its **stream_seq** stripmined to obtain the following

```
kernel (n)
  (λi li → stream_seq
              (λq a (ps': [q]int) →
                stream_seq (f i) (a) ps')
              (v)
              ps)
```

This kernel exhibits an optimization opportunity for the streamed array `ps`, as it is invariant to the first (and only) dimension. Concretely, the threads within a workgroup can collectively copy the chunk `ps'` into local memory, which we write as

```
kernel (n)
  (λi li → stream_seq
             (λq a (ps': [q]int) →
               let x = ps'[li]
               let ps'' = local q x
               in stream_seq (f i) (a) ps'')
             (v)
             ps)
```

Recall that `li` is the local thread index. Then, `ps''` is a local memory array created by collective copying (**`local`**), and used instead of `ps'` in `f`. The size `q` is then bounded (at runtime) by the group size used for this kernel, such that the array `ps''` will fit in local memory. For simplicity, we assume that the size of array `ps` is divisible by some reasonable group size. The trick is that every thread within a workgroup reads a distinct element from `ps'`, by using the local thread index. The array `ps''` is visible by all threads within the workgroup, so **`local`** in effect constitutes intra-group communication.

When tiling along a dimension *i* (1 in the above example), it must be ensured that each consecutive `q` threads along that dimension belong to the same GPU group. In general, this is done by forcing the group size to unit along all non-tiled dimensions, although this is not necessary in the case of a unidimensional kernel.

One-dimensional tiling is used for the *n*-body benchmark discussed in Section 10.1.1. Note that the dimensionality of the kernel need not be restricted to one for one-dimensional tiling to apply. Section 9.2.3 contains an example of one-dimensional tiling applied in a two-dimensional kernel.

9.2.2 Two-Dimensional Tiling

Futhark also supports two-dimensional tiling, where two streamed arrays are invariant to different parallel dimensions. A matrix multiplication that exhibits the pattern appears as follows:

```
let yss' = rearrange (1,0) yss
in kernel (n,l) (λi j li lj →
     let xs = xss[i]
     let ys = yss'[j]
     in stream_seq f (0) xs ys)

where f = λc acc xs' ys' →
          loop (acc) for i < c do
            acc + xs'[i] * ys'[i]
```

The chunk function `f` computes the dot product of its two input arrays and adds

it to the accumulator. Prior to kernel extraction, the **stream_seq** was in the form of a **redomap**. The arrays `xs` and `ys` are both invariant to at least one dimension of the kernel, so they can be tiled as follows:

```
let yss' = rearrange (1,0) yss
in kernel (n,l) (λi j li lj →
     let xs = xss[i]
     let ys = yss'[j]
     in stream_seq (g li lj) (0) xs ys

where g = λli lj q acc xs' ys' →
          let x = xs'[lj]
          let y = ys'[li]
          let xss' = local q q x
          let xs'' = xss'[li]
          let yss' = local q q y
          let yss_tr = rearrange (1,0) yss'
          let ys''   = yss_tr[lj]
          in stream_seq f acc xs'' ys''
      f = as before
```

Operationally, the **local** expressions creates a two-dimensional array in local memory, which is then immediately indexed with the local thread index along dimension 1 (or 2), resulting in a one-dimensional array. The core IR used in this thesis does not support indexing along any but the first dimension; instead we transpose before indexing.

9.2.3 A Complicated Instance of Tiling

The LavaMD benchmark from Section 10.1.1 exhibits an interesting tiling pattern, in which the to-be-tiled array is the result of an indirect index computed by a function `f`, all nested inside of a sequential loop. A simplified reproduction follows.

```
kernel (n,m) (λi j li lj →
  loop (outer_acc) = (...) for l < k do
    let i' = f l i
    let xs = xss[i']
    in stream_seq (h j) outer_acc xs)

where f l i = let p = if 0 < l then ps[l,i] else j
              in js[p]
      h j = some function
```

Since the computation of the array `xss` is invariant to the first dimension of the kernel (of size `n`), it can be tiled as follows:

```
kernel (n,m) (λi j li lj →
  loop (outer_acc) = (...) for l < k do
    let i' = f l i
    let xs = xss[i']
    in stream_seq (g j lj) (0) outer_acc xs)

where f l i = let p = if 0 < l then ps[l,i] else i
              in js[p]
      g = λj lj q acc xs' →
              let x = xs'[lj]
              let xss' = local 1 q x
              let xs'' = xss'[0]
              in stream_seq (h j) acc xs''
      h j = some function
```

Note that the tiling operation itself is not affected at all by the convoluted computation of the index `j'`. All that we need is the ability to compute which kernel dimensions `j'` is invariant to, which is reasonably simple in a pure language such as Futhark.

Again we assume that the second dimension (of size `m`) can be divided by a reasonable group size (say, 256). This assumption can be removed by the addition of various tedious bounds checks and branches. The use of **`local`** `1 q` forces that the two-dimensional group size is always 1 along the outermost dimension, which maximises the ratio of local-to-global memory accesses for array `xs`.

9.3 Related Work

The intent of this chapter is not to argue that Futhark's scheme for tiling and achieving coalesced memory accesses is superior to other techniques. Rather, we have merely shown that the moderate flattening algorithm does not actively *prevent* us from performing such optimisations, as is the case for full flattening.

For example, in comparison to Futhark, imperative analyses [Yan+10; Ver+13] are superior at performing all kinds of tiling, for example hexagonal time tilling [Gro+14] and achieving memory coalescing by semantically transposing arrays on the fly (via tiling). However, non-affine array indexes may restrict applicability: for example, indirect-array accesses would prevent traditional analyses from optimising memory coalescing for the OptionPricing benchmark (Section 10.1.3), where Futhark's simpler, transposition-based approach succeeds.

Chapter 10

Empirical Validation

When a new programming model is proposed, it must be demonstrated that interesting programs can be written within the model. While the model espoused by this thesis—functional array programming—is not new, our claim that it is a good foundation for obtaining parallel performance requires evidence.

This chapter serves as an empirical validation of the compilation strategy discussed in this thesis, and the practical usability of the Futhark compiler as a whole (the *fifth contribution* from the introduction). We discuss several programs implemented in Futhark, with a focus on their runtime performance. While we cannot claim that the current Futhark compiler embodies the full potential of functional array programming, the results presented here constitute at least a lower bound on that potential.

The chapter is divided into two main parts: first we have manually translated a range of benchmark programs from other languages or libraries to Futhark (Section 10.1). Second, we perform a deeper analysis of the performance of an irregular program that, due to dataset-sensitivity, is not as easy to benchmark (Section 10.2). We will use the term *reference implementation* to refer to programs that were not written by us, and *Futhark implementation* to refer to our translated Futhark programs. Any Futhark code examples will use the source language, not the IR from Section 5.2 that was used to discuss the design of the compiler.

10.1 Empirical Validation in Bulk

Most of the programs presented in this section are from the published benchmark suites Rodinia 3.1 [Che+09], Parboil 2.5 [Str+12], and FinPar [And+16]. The comparison with hand-written implementations is to show the potential cost of using a high-level language. However, as we shall see, even published code often contains significant inefficiencies, which occasionally leads to Futhark outperforming the reference implementations. A handful of benchmarks are from Accelerate 1.0 [McD+13],

		Runtime in milliseconds			
		NVIDIA K40		AMD W8100	
Benchmark	**Dataset**	**Ref.**	**Fut.**	**Ref.**	**Fut.**
Rodinia					
Backprop	Input layer size equal to 2^{20}	52.2	21.2	233.2	9.8
CFD	fvcorr.domn.193K	2893.6	3487.5	1637.8	1955.5
HotSpot	1024×1024; 360 iterations	51.6	60.3	3178.4	54.4
k-means	kdd_cup	1680.3	664.2	1377.6	1264.4
LavaMD	`boxes1d` = 10	7.1	9.4	5.8	7.4
LUD	2048	40.6	111.1	28.7	132.2
Myocyte	`workload` = 2^{16}; `xmax` = 3	3881.5	806.2	—	1621.0
NN	Default duplicated 20 times	182.9	13.4	331.0	60.3
Pathfinder	Array of size 10^5	26.1	9.4	96.3	2.7
SRAD	502×458; 100 iterations	21.8	22.1	49.9	41.8
Parboil					
MRI-Q	large	27.6	22.8	15.6	20.1
SGEMM	medium	3.4	7.2	3.2	4.7
Stencil	default	179.0	253.7	97.8	223.7
TPACF	medium	655.3	873.4	272.0	557.9
FinPar					
LocVolCalib	small	495.8	281.4	442.8	248.1
	medium	319.9	190.0	214.0	139.7
	large	1484.2	2038.5	1678.1	3279.9
OptionPricing	small	4.7	7.3	4.2	6.1
	medium	10.4	17.7	6.7	13.6
	large	97.1	158.6	83.2	175.3
Accelerate					
Crystal	Size 2000, degree 50	27.9	11.2	—	10.3
Fluid	3000×3000; 20 iterations	15.0	19.9	—	14.5
Mandelbrot	4000×4000; 255 limit	6.1	6.0	—	5.0
N-body	$N = 10^5$	559.0	127.6	—	141.7
Tunnel	4000×4000	94.2	72.5	—	89.8

Table 4: Benchmark datasets and average runtimes, computed over ten executions of every benchmark. See Figures 61 to 64 for a visualisation of the results. Missing entries indicate that a benchmark implementation was not runnable for a given configuration.

a Haskell eDSL for data-parallel array programming. These are included to show the performance of Futhark compared to an existing mature GPU language.

All programs have been manually ported to Futhark, and compiled and run with the default settings. We show how the performance of the Futhark code compares to the performance of the original reference implementations. In some cases, we also discuss the programming techniques used to obtain efficient Futhark implementations. For most benchmarks, we use only a single dataset, as the relative performance for these is not dataset-sensitive (except for pathological cases).

While some of the benchmarks are small, others are decidedly nontrivial. For example, Stencil from Parboil is implemented as 13 non-blank non-comment lines of Futhark, while Myocyte from Rodinia requires 883. The total number of non-blank non-comment lines in the Futhark implementations is 2984, with a median of 98.

We have evaluated the performance of the generated code on two different test systems:

1. An NVIDIA K40 GPU installed in a server-grade system with an Intel Xeon E5-2560 CPU and 128GiB of RAM.

2. An AMD W8100 GPU installed in a desktop-class system with an Intel Core 2 Quad CPU and 4GiB of RAM.

The intent is to show that the OpenCL code generated by the Futhark compiler works on both NVIDIA and AMD GPUs. However, note that the two systems are very dissimilar—the AMD GPU is installed in a rather old computer, and it is possible that the slow CPU and anaemic amount of system memory may bottleneck the GPU[1]. This may explain some of the unusual behaviour we will see on the AMD GPU, particularly for reference implementations. In particular, three reference implementations from Rodinia—Backprop, Hotspot, and Pathfinder—exhibit extreme and inexplicable slowdown on the AMD GPU. We consider these results anomalous, and will discard from from further discussion (although they are included in the graphs and tables). Ideally, the NVIDIA and AMD GPUs would be installed in systems that are otherwise similar, but such were not available for the writing of this thesis.

CUDA, as an API proprietary to NVIDIA, can be executed only on the NVIDIA K40 GPU. Thus, when selecting reference implementations, we have picked those written in OpenCL, to allow us to run the same code on both platforms. Reference implementations in OpenCL were not available for all benchmarks; these will be discussed on a case-by-case basis. All Futhark implementations executed successfully on both test systems.

We invoke the Futhark compiler with no benchmark-specific flags—no tuning is done, and the exact same code is executed on both of our test systems. Informal investigation suggests that approximately 10% speedup could be obtained by tweaking GPU group sizes.

[1] Unusually, the GPU actually has *more* memory than the CPU; a reversal of the usual situation.

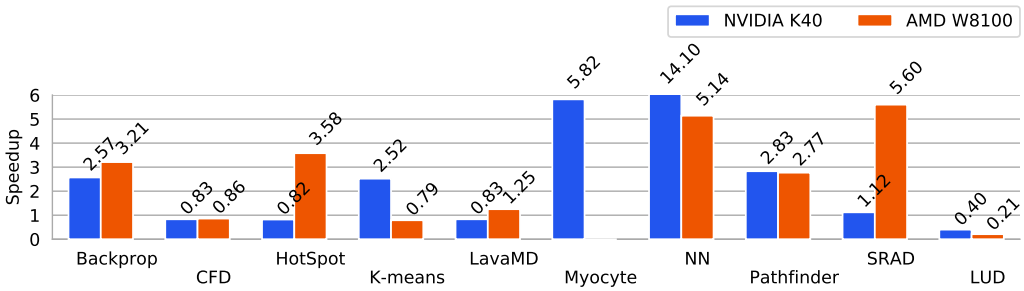


Figure 61: Relative speedup compared to reference implementations for Rodinia (the bar for NN is truncated for space reasons).

The compiler inserts instrumentation that records total runtime minus the time taken for (i) loading program input onto the GPU, (ii) reading *final* results back from the GPU, and (iii) OpenCL context creation and kernel build time. Excluding these overheads emphasises the performance differences. Any other host-device communication/copying is measured. When necessary, the reference implementations have been modified to time similarly. When in doubt, we have erred in favour of the reference implementations.

The results are shown on Table 4, and discussed in greater detail below. Most of our slowdown is related to generic issues of unnecessary copying and missing micro-optimisation that are common to compilers for high-level languages. Section 10.1.5 contains a brief discussion of how the major optimisations performed by the Futhark compiler affect the performance of the benchmarks.

Futhark implementations of all benchmarks discussed here, as well as several more, are publicly available at the following URL:

```
https://github.com/diku-dk/futhark-benchmarks
```

10.1.1 Ten Benchmarks from Rodinia

Rodinia is a popular benchmark suite that contains approximately 21 benchmarks. Of these, we have selected those 10 benchmarks that look the most friendly from a data-parallel perspective, and were expressible in terms of a nested composition of **`map`**, **`reduce`**, **`scan`**, **`stream_red`**, **`stream_map`**.

There is great variation in the performance of the Rodinia reference implementations. Some, such as Myocyte or NN, contain oversights that negatively affect GPU performance. Others, such as LUD or LavaMD, are based on clever algorithms and carefully implemented. Some Rodinia implementations exhibit massive slowdown on the AMD GPU for reasons that we cannot determine. We conjecture that this is related to the under-powered CPU on the machine hosting the AMD GPU, but this is only a suspicion. The speedups are shown on Figure 61.

The speedup on Backprop seems related to a reduction that Rodinia has left se-

quential. Running time of the training phase is roughly equal in Rodinia and Futhark ($\sim 10\,ms$).

Rodinia's implementation of HotSpot, a two-dimensional stencil code, uses time tiling [Gro+14], which seems to pay off on the NVIDIA GPU, but not on AMD. On the NVIDIA GPU, the majority of Futhark's slowdown is due to how memory management is done for the arrays involved in the time series loop in the stencil. At a high level, the stencil is a sequential loop containing a nested **map** over the $m \times m$ iteration space:

```
loop s = s0 for i < n do
  let s' = map (λi → map (f s i) [0...m-1]) [0...m-1]
  in s'
```

Two distinct memory blocks are necessary, because several elements of `s` are combined to construct one element of `s'`. The reference implementation uses two pre-allocated buffers, and switches between them by swapping the pointers. This is a common technique for implementing stencils. Instead of swapping pointers, The Futhark compiler copies the entire intermediate result instead, and these copies account for 30% of runtime.

Our speedup on k-means is due to Rodinia not parallelising computation of the new cluster centres, which is semantically a histogram, which can be implemented as a segmented reduction.

The default dataset for the Myocyte benchmark was expanded because its degree of parallelism was one (`workload` = 1). We used Rodinia's CUDA implementation rather than its OpenCL implementation, as the latter is not fully parallelised. We attribute our speedup to automatic coalescing optimisations, which is tedious to do by hand on such large programs.

Our speedup on NN is due to Rodinia leaving 100 **`reduce`** operations for finding the nearest neighbours sequential on the CPU. This is possibly because the reduce operator is atypical: it computes both the minimal value and the corresponding index, much like the example in Section 2.2.1. Speedup is less impressive on the AMD GPU, due to higher kernel launch overhead—this benchmark is dominated by frequent launches of short kernels.

For Pathfinder, Rodinia uses time tiling, which, unlike HotSpot, does not seem to pay off on the tested hardware.

The reference implementation of LUD uses a clever block-based algorithm that makes efficient use of local memory on the GPU. The Futhark implementation is similar, but the Futhark compiler is not yet able to map it as efficiently to the GPU hardware. The LUD algorithm uses block decomposition, and the block size (not to be confused with the CUDA term "thread block size", which is something else) is a programmer-given constant. Both the reference and Futhark implementation simply hard-code a constant that appears to give good performance in practice: 32

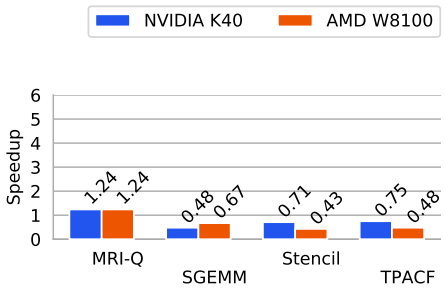


Figure 62: Relative speedup compared to reference implementations for Parboil.

for Futhark, and 16 for the reference implementation.

10.1.2 Four Benchmarks from Parboil

The Parboil benchmark suite is smaller than Rodinia, containing 11 benchmarks, but the implementations are generally of higher quality. Furthermore, Parboil comes with excellent built-in support for instrumentation and validation of results. The Parboil benchmarks tend to be more difficult than those found in Rodinia, so we have only ported four of them to Futhark. The speedups are shown on Figure 62.

MRI-Q is semantically an outer **`map`** surrounding an inner **`map`**-**`reduce`** operation on an array that is invariant to the outer **`map`**. The Parboil implementation is based on heavily unrolling the inner loop, while the Futhark compiler performs one-dimensional tiling in local memory (Section 9.2.1). This appears to be the superior strategy on the NVIDIA GPU, but not the AMD GPU.

SGEMM is an implementation of a common matrix primitive that computes

$$C \leftarrow \alpha \times A \times B + \beta \times C$$

where A, B, C are matrices and α, β are scalars. The Futhark compiler performs two-dimensional tiling (Section 9.2.2) in local memory, while the Parboil implementation performs more sophisticated *register tiling*. SGEMM is a well-studied primitive, and it is hard for a compiler to match a hand-tuned implementation.

Stencil is a straightforward stencil code. The Futhark implementation suffers due to poor memory management that induces extra copies, as with Rodinia's HotSpot.

TPACF is semantically a histogram, which Parboil implements using clever use of local memory. In Futhark, we implement it using a segmented reduction, which is not as efficient.

10.1.3 Two Benchmarks from FinPar

FinPar is a small benchmark suite translated from real-world financial kernels. The problems are sophisticated, with nontrivial use of nested parallelism, and the reference implementations are well implemented. We have ported only two out of the

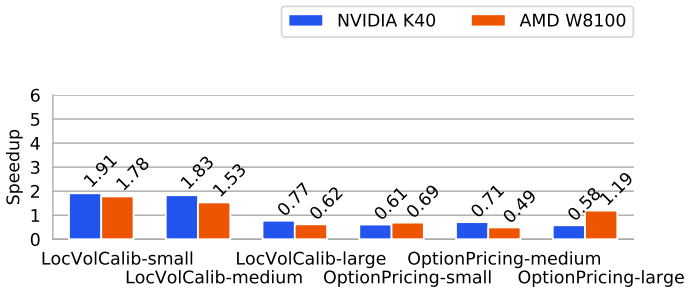


Figure 63: Relative speedup compared to reference implementations for FinPar. Each benchmark is applied to three different data sets.

three FinPar benchmarks to Futhark, as the third one (InterestCalib) contains (limited) irregular parallelism. Due to the small size of FinPar, we have evaluated each benchmark on all three available data sets. The speedups are shown on Figure 63.

OptionPricing is essentially a **`map`**-**`reduce`**-composition. The benchmark primarily measures how well the Futhark compiler sequentialises excess parallelism inside the complex **`map`** function. We see approximately equal relative performance for each of the three data sets.

LocVolCalib from FinPar is an outer **`map`** containing a sequential **`for`**-loop, which itself contains several more **`map`**s. Exploiting all parallelism requires the compiler to interchange the outer **`map`** and the sequential loop. The slowdown on the AMD GPU is due to transpositions, inserted to fix coalescing, being relatively slower than on the NVIDIA GPU.

The *small* and *medium* datasets for LocVolCalib have the interesting property that they do not provide enough parallelism in the outer loops to fully saturate the GPU. Note the absolute runtime numbers on Table 4, where the runtime for the *small* dataset is greater than that for the *medium* dataset, despite the latter actually requiring more work. Two distinct LocVolCalib implementations are provided by FinPar: one that exploits only outer-level parallelism, and one that exploits *all* available parallelism. The former is the right choice for the *large* dataset, and the one we compare against here, as it is also roughly how the current Futhark compiler parallelises LocVolCalib. However, the FinPar implementation that fully exploits all parallelism outperforms Futhark on the small and medium datasets (not shown here). Handling cases such as this transparently is the main motivation for multi-versioned code (Section 8.3).

10.1.4 Five Benchmarks from Accelerate

Accelerate is a mature Haskell-embedded language for data-parallel array programming. The benchmarks picked here are presented as “examples”, and are not part of a benchmark suite as such. While they are written by the Accelerate authors

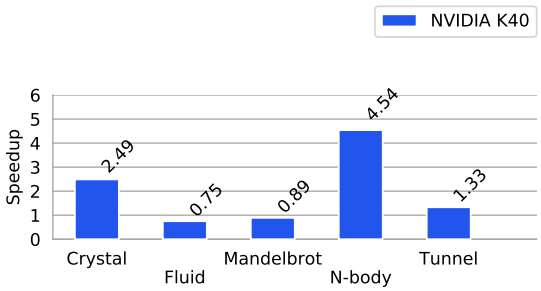


Figure 64: Relative speedup compared to reference implementations for Accelerate.

themselves, and contain built-in support for benchmarking, it is possible that performance has been sacrificed in the interest of readability. We use the recently released `llvm-ptx` GPU backend, which performs significantly better than the old `cuda` backend. Unfortunately, this backend is still NVIDIA-specific, so we were unable to execute Accelerate on the AMD GPU.

The benchmarks were all straightforward to translate to Futhark, as Accelerate does not support nested parallelism or low-level GPU programming. One major and interesting source of performance differences is that Accelerate is a Just-In-Time (JIT) compiler for an embedded language, while Futhark is a conventional Ahead-Of-Time (AOT) compiler. As a result, Accelerate has access to dynamic information that can be used to perform code specialisation, but its embedded nature limits the compilers ability to see the program as a whole. On the other hand, the Futhark compiler can perform more global optimisations, but cannot inline dataset-specific constants for loop iteration counts and similar. The trade-offs between JIT and AOT compilation are a fascinating and deep subject that we will not delve further into here, but is certainly worthy of further study. Speedups are shown on Figure 64.

Crystal is a straightforward **`map`** containing an inner **`map`**-**`reduce`** over a small array. It is unclear why Futhark outperforms Accelerate on this benchmark.

Fluid is a two-dimensional stencil code with complicated edge conditions. As with Rodinia's HotSpot, Futhark's approach to memory management causes unnecessary copies for each of the outer sequential loops, giving Accelerate the edge.

The Mandelbrot benchmark is a straightforward **`map`** containing a **`while`** loop, with little opportunity for optimisation.

N-body is operationally a **`map`** containing an inner **`map`**-**`reduce`** over an array invariant to the outer array. The Futhark compiler performs one-dimensional tiling in local memory to optimise access to this array. Accelerate does not perform tiling, which gives the edge to Futhark.

Tunnel is similar to Crystal, and contains a straightforward **`map`** containing an inner **`map`**-**`reduce`** over a small array. Performance is similar between Futhark and Accelerate.

10.1.5 Impact of Optimisations

Impact was measured by turning individual optimisations off and re-running benchmarks on the NVIDIA GPU. We report only where the impact is non-negligible.

Fusion (Chapter 7) has a significant impact on k-means (×1.42), LavaMD (×4.55), Myocyte (×1.66), SRAD (×1.21), Crystal (×10.1), and LocVolCalib (×9.4). Without fusion, OptionPricing, N-body, MRI-Q, and SGEMM fail due to increased storage requirements.

In the absence of in-place updates, we would have to implement k-means as on Figure 7—the resulting program is slower by ×8.3. Likewise, LocVolCalib would have to implement its central `tridag` procedure via a less efficient **`scan`**-**`map`** composition, causing a ×1.7 slowdown. OptionPricing uses an inherently sequential Brownian Bridge computation that is not expressible without in-place updates.

The coalescing-by-transposition transformation (Section 9.1) has a significant impact on k-means (×9.26), Myocyte (×4.2), OptionPricing (×8.79), and LocVolCalib (×8.4). Loop tiling (Section 9.2) has an impact on LavaMD (×1.35), MRI-Q (×1.33), SGEMM (×2.3), N-body (×2.29), and LUD (×1.10).

10.2 Benchmarking an Implementation of Breadth-First-Search

The Rodinia benchmark suite contains a benchmark, BFS, that implements breadth-first search. This problem is highly irregular, and its performance is very dataset-sensitive. As a result, we have excluded it from the presentation of the general benchmark results, and instead dedicated this section to discuss the issues. The following serves as an example of benchmarks where measuring performance is not quite straightforward, and incidentally as an example of how to implement irregular problems in Futhark.

Figure 65 shows an excerpt of Rodinia's imperative-but-parallel implementation of the breadth-first search algorithm, which traverses the graph and records in the `cost` array the breadth level of each graph node. The code excerpt processes in parallel all the nodes, indexed by `src_id`, on the current breadth level (i.e., the ones with the `mask[src_id]` set). For each such node, the breadth level of all its unvisited neighbours (i.e., `!visited[id]`) are updated to `cost[src_id]+1` in the sequential (inner) loop at line 11. The problem with this code is that the update of `cost` potentially introduces data races in the case when two nodes on the same level are connected to a common (unvisited) node. What makes it still safe is the property that the output dependences are idempotent (i.e., the updated value is the same across different outermost iterations).

Figure 66 shows a simple, albeit work-inefficient, translation to Futhark of the discussed imperative code, which uses padding to satisfy array regularity. First, the indices of the active nodes (i.e., the ones on the current breadth level) are identified by the **`filter`** operation on line 8. This contributes significantly to final perfor-

mance. Second, the maximal number of edges of these nodes, denoted by `e_max`, is computed via a **`map`**–**`reduce`** composition on lines 10–12. Third, the **`map`** nest computes two two-dimensional arrays of innermost size equal to `e_max`, in which the first one corresponds to the indices of the unvisited neighbours, and the second one to their breadth level (`costs`). The indices that are outside the edge degree of a node are marked with out-of-bounds indices (`-1`), and similarly for the nodes that were already visited. Fourth, the index-value arrays are flattened (at lines 29-30) and the **`scatter`** bulk operator is used to update the `cost` and `updating_mask` arrays at lines 32–37.

Table 5 shows the running time of Futhark and Rodinia implementations on four datasets. The Rodinia implementation exploits only the parallelism of the outer loop, and it wins on the datasets in which the graph is large enough to fully utilise the hardware, while Futhark exploits both levels of parallelism and wins on smaller graphs with large edge degree. Interestingly, this is not simply a question of how much to parallelise, but a fundamental algorithmic design decision. The unsolved challenge is a representation and technique that can fuse the Futhark code in Figure 66 into something resembling the one-level-parallel C code in Figure 65, and furthermore generate both versions of the code.

```
1  // n is the number of graph nodes
2  for (int src_id = 0; src_id < n; src_id++ ) { // parallel
3      if (mask[src_id] == true) {
4          mask[src_id] = false;
5          for (int i = nodes[src_id].starting; // sequential
6               i < (nodes[src_id].num_edges
7                    +nodes[src_id].starting);
8               i++) {
9              int dst_id = edges_dest[i];
10             if(!visited[dst_id]) {
11                 cost[dst_id] = cost[src_id] + 1;
12                 updating_mask[dst_id] = true;
13             }
14         }
15     }
16 }
```

Figure 65: Imperative code for breadth-first search.

Finally, we remark that the presented Futhark code is work-inefficient due to the "padding" of the edge-degree of a node, but the **`scatter`** construct also enables a work-efficient implementation (not shown), which corresponds to full flattening. In our tests, this is slower than the padded one due to the overhead of multiple scans.

```
let step [n][e] (cost: [n]i32)
                (nodes_starting: [n]i32)
                (nodes_num_edges: [n]i32)
                (edges_dest: [e]i32)
                (visited: [n]bool)
                (mask: [n]bool)
    : ([n]i32, [n]bool, []i32) =
  let active_indices = filter (λi → mask[i]) (iota n)
  let n_indices = length active_indices
  let e_max =
    reduce i32.max 0
    (map (λi → nodes_n_edges[i]) active_indices)
  let (node_ids_2d, costs_2d) =
    map (λsrc_id: ([e_max]i32, [e_max]i32)  →
          let s_index = nodes_starting [src_id]
          let n_edges = nodes_num_edges[src_id]
          let edge_indices = map (+s_index) (iota e_max)
          let node_ids =
            map (λi →
                  if i < s_index + n_edges
                  then let dst_id = edges_dest[i]
                       in  if !visited[dst_id]
                           then dst_id else -1
                  else -1)
                edge_indices
          let costs = replicate e_max (cost[src_id] + 1)
          in (node_ids, costs))
        active_indices
  let node_ids = reshape flat_len node_ids_2d
  let costs = reshape flat_len costs_2d
  let flat_len = e_max * n_indices
  let mask' =
    scatter mask active_indices (replicate n_indices false)
  let cost' =
    scatter cost node_ids costs
  let updating_mask' =
    scatter updating_mask node_ids (replicate flat_len true)
  in (mask', cost', updating_mask')
```

Figure 66: Work-inefficient Futhark code for breadth-first search.

Dataset	Version	Runtime	Speedup
2000 nodes, 1000 edges each	Futhark	4.0ms	×1.9
	Rodinia	7.6ms	
1000 nodes, 10–1000 edges each	Futhark	1.7ms	×3.7
	Rodinia	6.3ms	
100,000 nodes, 6 edges each	Futhark	6.7ms	×0.25
	Rodinia	1.7ms	
100,000 nodes, 10–500 edges each	Futhark	153.1ms	×0.43
	Rodinia	66.5ms	

Table 5: Performance of Rodinia and Futhark breadth-first search implementations on various datasets. Executed on an NVIDIA GTX 780 Ti with random graphs (uniform distribution). Reported runtime is average over 10 runs.

Part III

Closing Credits

Chapter 11

Conclusions and Future Work

We have presented a fully automatic optimizing compiler for Futhark, a pure functional array language. Futhark is a simple language that supports just a few core concepts, yet is more expressive than prior parallel languages of similar performance, in particular by supporting nested parallelism. While more flexible parallel languages exist (notably NESL), these have not yet been shown to obtain good GPU performance in practice.

We have demonstrated a combination of imperative and functional concepts by supporting in-place updates with safety guaranteed by uniqueness types rather than complicated index-based analysis. We support not just efficient top-level imperative code with in-place updates, but also sequential code nested inside parallel constructs, all without violating the functional properties on which we depend for safe parallel execution.

We have also shown how size-dependent array types can be inferred from a size-agnostic source program, via a combination of type-driven rules and function slicing. Our slicing technique results in neglible overhead in practice.

By introducing novel parallel streaming constructs, we provide better support for efficient sequential execution of excess parallelism than is possible with the classical parallel combinators. We have also shown how these constructs permit highly general fusion rules, in particular permitting sequential fusion without losing the potential for parallel execution.

All of these build up to our primary result, the moderate flattening algorithm, which allows the efficient exploitation of easily accessible "common case" parallelism. The moderate flattening algorithm sequentialises excess parallelism while keeping intact the high-level invariants provided by the original parallel and purely functional formulation, which permits further locality-of-reference optimisations. We have demonstrated this capability by showing how to automatically repair some cases of non-coalesced memory accesses, as well as performing simple block tiling of loops. We argue that the moderate flattening algorithm can be developed further into a *gradual flattening algorithm*, which uses multi-versioned code to exploit a varying

amount of parallelism of the program, dependent on the characteristics of the input data encountered at runtime.

To support our claims, we have validated our approach on 21 benchmark programs, which are compiled to GPU code via OpenCL. Compared to reference implementations, the performance ranges from $\times 0.21$ slowdown to $\times 13$ speedup, and is competitive on average. Our results show that while the ease of high-level structural transformation permitted by a functional language is powerful, attention must still be paid to low-level issues such as communication costs and memory access patterns.

We have made the developed compiler and all benchmark programs freely available for reproduction, study, or further development.

11.1 Limitations and Future Work

The primary limitation of Futhark as a language is the lack of support for irregular arrays. Some problems, such as graph algorithms, are naturally irregular, and transforming them to a regular formulation is tedious and error-prone. Worse, such transformation is essentially manual flattening, which tends to result in code that is hard for the compiler to analyse and optimise. It remains to be seen how we can either modify the algorithms in question to exhibit less irregularity, or extend Futhark with lightweight support for some cases of irregular parallelism, without sacrificing the ability of the compiler to generate efficient code. After all, Futhark's entire *raison d'être* is performance—there are already many functional languages that are far more expressive, so improving flexibility at great cost in runtime performance is not a goal.

However, even with the current language, irregularity still rears its ugly head for the compiler. It is not hard to write a program that, although it uses only regular arrays, implicitly expresses irregular *parallelism*, which cannot in general be handled by our current approach. The reason is that we expect to be able to pre-allocate memory before entering GPU kernels, but it is not hard to write a program in which the size of intermediate arrays is thread-variant. For example, consider the following expression.

```
map (λi → reduce (+) 0 (iota i)) is
```

The size of the array produced by `iota i` may differ for each element of the array `is`, which means the total memory requirement of the **`map`** cannot be immediately known. The problem has multiple solutions. In the most general case, we can apply full flattening—this removes all forms of irregularity, but the overhead can be significant. In other cases, a slicing approach can be used. For example, we can precompute the total memory required, allocate one large slab of memory, and use a **`scan`** to compute an offset for each iteration of the **`map`**:

```
let os = scan (+) 0 is
in map (λ(i,o) →
          -- iota i located at offset 'o'
          -- in some memory block.
          reduce (+) 0 (iota i))
       (zip is os)
```

This is not a language issue—our sequential compiler pipeline is able to compile any valid Futhark program to sequential code—but a limitation in our compilation strategy for parallel code. Our strategy is to focus on development a technique for exploiting "common case" parallelism efficiently, and only later extend it to support more exotic cases. This work will take the form of a *gradual flattening* algorithm that employes multi-versioned code as discussed in Section 8.3.

Various other opportunities for improvements have been noted throughout the thesis. These are summarised below.

Provide a language mechanism for reasoning about irregularity: As discussed above, irregularity is one of the main weak points of Futhark's approach. Even when we add support for irregular parallelism, it is likely that irregular programs will run slower than corresponding regular programs. It would be useful to come up with a language-based model for reasoning about regularity of parallelism, in the same way that size annotations already let us reason about the regularity of arrays.

Support tail-recursion directly: Futhark does not directly support tail-recursion, which is mostly due to concerns about implementation simplicity. Instead, the programmer is required to manually express the tail-recursive functions using the **`loop`** syntax. It would be useful if the Futhark compiler could automatically perform this conversion, and report an error on non-tail recursive functions. Ideally, this would also support mutually tail-recursive functions.

Support higher-order functions: While higher-order functions can be imitated using the module system, as shown on Figure 4, this is rather verbose. It would be useful if higher-order functions could be used directly, with type rules guaranteeing that they can be efficiently compiled away early in the compilation process, using a defunctionalisation transformation. We believe that it can be guaranteed that the result of such a transformation is efficient (contains no branches or indirect accesses) if a function is never used as a **`loop`** parameter, returned from a branch, or put into an array. However, it is not yet clear how higher-order functions would interact with size annotations, or with the uniqueness type system.

Iteration analysis when coalescing memory accesses: Section 9.1 describes a transformation that changes the in-memory representation of arrays in order to en-

sure that they will be accessed efficiently by the GPU. However, it relies on the simplifying assumption that if an individual GPU thread takes a slice of an array, then that slice will be traversed sequentially. This is not necessarily true. The compiler could instead perform an actual analysis to determine how the thread accesses memory, and represent the array in exactly the form that leads to coalesced memory access patterns. Since the sequential looping operations performed by the thread are (potentially) still represented using SOACs, the compiler has enough high-level information available to perform the analysis without getting bogged down in the complicated index analysis discussed in Section 2.7.

Improving memory management for stencils: As discussed in Section 10.1.1, the memory management strategy used by the Futhark compiler exhibits pathological behaviour for stencil programs (or more generally, any program with an outer sequential convergence loop), in which an extra copy of the result is performed. This is an artifact of the *double buffering* transformation that the Futhark compiler performs in order to move a memory allocation out of the loop. This transformation should be improved to perform pointer swapping in the conventional way, rather than copying the result.

Parallel operators for certain irregular problems: Section 10.2 shows an irregular problem (breadth-first search) in which the parallel operators provided by Futhark are insufficient for permitting the compiler to generate code similar to what a human would write by hand. It is not clear whether a new source-language feature is required, or whether we could simply augment the core language with a new "streaming **`scatter`**" construct that is produced by fusion.

As the Futhark compiler is publicly available free software developed in the open, we naturally encourage all interested readers to contribute in solving the above (and other) problems:

`https://github.com/diku-dk/futhark`

Happy hacking!

Bibliography

[17] *CUDA API Reference Manual*. 8.0. NVIDIA. Oct. 2017. URL: http://docs.nvidia.com/cuda.

[Aho+07] AHO, ALFRED V., MONICA S. LAM, RAVI SETHI, and JEFFREY D. ULLMAN. *Compilers, Principles, Techniques, and Tools*. Pearson Addison Wesley, 2007. ISBN: 0-321-49169-6.

[And+16] ANDREETTA, CHRISTIAN, VIVIEN BÉGOT, JOST BERTHOLD, MARTIN ELSMAN, FRITZ HENGLEIN, TROELS HENRIKSEN, MAJ-BRITT NORDFANG, and COSMIN E. OANCEA. "FinPar: A Parallel Financial Benchmark". In: *ACM Trans. Archit. Code Optim.* 13.2 (June 2016), 18:1–18:27. ISSN: 1544-3566.

[Bag+15] BAGHDADI, RIYADH, ULYSSE BEAUGNON, ALBERT COHEN, TOBIAS GROSSER, MICHAEL KRUSE, CHANDAN REDDY, SVEN VERDOOLAEGE, ADAM BETTS, ALASTAIR F DONALDSON, JEROEN KETEMA, et al. "PENCIL: a platform-neutral compute intermediate language for accelerator programming". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 138–149.

[BD09] BOVE, ANA and PETER DYBJER. "Dependent types at work". In: *Language engineering and rigorous software development*. Springer, 2009, pp. 57–99.

[Ber+10] BERGSTRA, JAMES, OLIVIER BREULEUX, FRÉDÉRIC BASTIEN, PASCAL LAMBLIN, RAZVAN PASCANU, GUILLAUME DESJARDINS, JOSEPH TURIAN, DAVID WARDE-FARLEY, and YOSHUA BENGIO. "Theano: A CPU and GPU Math Compiler in Python". In: *Procs. of the 9th Python in Science Conference*. Ed. by WALT, STÉFAN van der and JARROD MILLMAN. 2010, pp. 3–10.

[Ber+13] BERGSTROM, LARS, MATTHEW FLUET, MIKE RAINEY, JOHN REPPY, STEPHEN ROSEN, and ADAM SHAW. "Data-only Flattening for Nested Data Parallelism". In: *Procs. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 81–92. ISBN: 978-1-4503-1922-5. DOI: http://dx.doi.org/10.1145/2442516.2442525. URL: http://doi.acm.org/10.1145/2442516.2442525.

[Ber+88] BERRY, M., D. CHEN, P. KOSS, D. KUCK, S. LO, Y. PANG, L. POINTER, R. ROLOFF, A. SAMEH, E. CLEMENTI, S. CHIN, D. SCHNEIDER, G. FOX, P. MESSINA, D. WALKER, C. HSIUNG, J. SCHWARZMEIER, K. LUE, S. ORSZAG, F. SEIDL, O. JOHNSON, and R. GOODRUM. "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers". In: *International Journal of Supercomputer Applications* 3 (1988), pp. 5–40.

[BF88] BRATLEY, PAUL and BENNETT L. FOX. "Algorithm 659 Implementing Sobol's Quasirandom Sequence Generator". In: *ACM Trans. on Math. Software (TOMS)* 14(1) (1988), pp. 88–100.

[BG95] BLELLOCH, GUY and JOHN GREINER. "Parallelism in Sequential Functional Languages". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: ACM, 1995, pp. 226–237. ISBN: 0-89791-719-7. DOI: http://dx.doi.org/10.1145/224164.224210. URL: http://doi.acm.org/10.1145/224164.224210.

[BG96] BLELLOCH, GUY E. and JOHN GREINER. "A Provable Time and Space Efficient Implementation of NESL". In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 213–225. ISBN: 0-89791-770-7. DOI: http://dx.doi.org/10.1145/232627.232650. URL: http://doi.acm.org/10.1145/232627.232650.

[Bir87] BIRD, R. S. "An Introduction to the Theory of Lists". In: *NATO Inst. on Logic of Progr. and Calculi of Discrete Design*. 1987, pp. 5–42.

[Bir89] BIRD, R. S. "Algebraic Identities for Program Calculation". In: *Computer Journal* 32.2 (1989), pp. 122–126.

[Ble+94] BLELLOCH, GUY E, JONATHAN C HARDWICK, JAY SIPELSTEIN, MARCO ZAGHA, and SIDDHARTHA CHATTERJEE. "Implementation of a Portable Nested Data-Parallel Language". In: *Journal of parallel and distributed computing* 21.1 (1994), pp. 4–14.

[Ble90] BLELLOCH, GUY E. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge, 1990.

[Ble96] BLELLOCH, GUY E. "Programming Parallel Algorithms". In: *Communications of the ACM (CACM)* 39.3 (1996), pp. 85–97.

[BR12] BERGSTROM, LARS and JOHN REPPY. "Nested Data-Parallelism on the GPU". In: *Procs. of Int. Conf. Funct. Prog. (ICFP)*. ACM. 2012, pp. 247–258.

[Bra13] BRADY, EDWIN. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593.

[Bro+16] BROWN, KEVIN J., HYOUKJOONG LEE, TIARK ROMPF, ARVIND K. SUJEETH, CHRISTOPHER DE SA, CHRISTOPHER ABERGER, and KUNLE OLUKOTUN. "Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO 2016. Barcelona, Spain: ACM, 2016, pp. 194–205. ISBN: 978-1-4503-3778-6. DOI: http://dx.doi.org/10.1145/2854038.2854042. URL: http://doi.acm.org/10.1145/2854038.2854042.

[BS93] BARENDSEN, ERIK and SJAAK SMETSERS. "Conventional and Uniqueness Typing in Graph Rewrite Systems". In: *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*. Vol. 761. LNCS. 1993, pp. 41–51.

[BS96] BARENDSEN, ERIK and SJAAK SMETSERS. "Uniqueness Typing for Functional Languages with Graph Rewriting Semantics". In: *Mathematical Structures in Computer Science* 6.6 (1996), pp. 579–612.

[BTV96] BIRKEDAL, LARS, MADS TOFTE, and MAGNUS VEJLSTRUP. "From Region Inference to von Neumann Machines via Region Representation Inference". In: *ACM Symposium on Principles of Programming Languages*. POPL'96. ACM Press, Jan. 1996, pp. 171–183.

[CGK11] CATANZARO, BRYAN, MICHAEL GARLAND, and KURT KEUTZER. "Copperhead: Compiling an Embedded Data Parallel Language". In: *Procs. of ACM Symp. on Principles and Practice of Parallel Programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 47–56. ISBN: 978-1-4503-0119-0. DOI: http://dx.doi.org/10.1145/1941553.1941562. URL: http://doi.acm.org/10.1145/1941553.1941562.

[Cha+07] CHAKRAVARTY, MANUEL M. T., ROMAN LESHCHINSKIY, SIMON PEYTON JONES, GABRIELE KELLER, and SIMON MARLOW. "Data Parallel Haskell: A Status Report". In: *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*. 2007, pp. 10–18.

[Che+09] CHE, S., M. BOYER, J. MENG, D. TARJAN, J. W. SHEAFFER, S. H. LEE, and K. SKADRON. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Oct. 2009, pp. 44–54. DOI: http://dx.doi.org/10.1109/IISWC.2009.5306797.

[Che77] CHEATHAM JR, THOMAS E. "Programming Language Design Issues". In: *Design and Implem. of Prog. Lang.* Springer, 1977, pp. 399–435.

[Chi+04] CHICHA, Y., M. LLOYD, C. OANCEA, and S. M. WATT. "Parametric Polymorphism for Computer Algebra Software Components". In: *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput.* Mirton Publishing House, 2004, pp. 119–130.

[CKF11] COLLOBERT, RONAN, KORAY KAVUKCUOGLU, and CLÉMENT FARABET. "Torch7: A Matlab-like Environment for Machine Learning". In: *BigLearn, Neural Information Processing Systems*. 2011.

[Col+14] COLLINS, ALEXANDER, DOMINIK GREWE, VINOD GROVER, SEAN LEE, and ADRIANA SUSNEA. "NOVA: A Functional Language for Data Parallelism". In: *Procs. of Int. Workshop on Libraries, Languages, and Compilers for Array Prog.* ARRAY'14. Edinburgh, United Kingdom: ACM, 2014, 8:8–8:13. ISBN: 978-1-4503-2937-8. DOI: http://dx.doi.org/10.1145/2627373.2627375. URL: http://doi.acm.org/10.1145/2627373.2627375.

[CSL07] COUTTS, DUNCAN, DON STEWART, and ROMAN LESHCHINSKIY. "Rewriting Haskell Strings". In: *Practical Aspects of Decl. Lang.* Springer, 2007, pp. 50–64.

[CSS12] CLAESSEN, KOEN, MARY SHEERAN, and BO JOEL SVENSSON. "Expressive Array Constructs in an Embedded GPU Kernel Programming Language". In: *Work. on Decl. Aspects of Multicore Prog DAMP*. 2012, pp. 21–30.

[CSS15] CHATARASI, PRASANTH, JUN SHIRAKO, and VIVEK SARKAR. "Polyhedral optimizations of explicitly parallel programs". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 213–226.

[DAS12] DUBOIS, MICHEL, MURALI ANNAVARAM, and PER STENSTRM. *Parallel Computer Organization and Design*. New York, NY, USA: Cambridge University Press, 2012. ISBN: 9780521886758.

[DT13] DIMARCO, JEFFREY and MICHELA TAUFER. "Performance impact of dynamic parallelism on different clustering algorithms". In: *Proc. SPIE*. Vol. 8752. 2013, 87520E.

[Dub+12] DUBACH, CHRISTOPHE, PERRY CHENG, RODRIC RABBAH, DAVID F. BACON, and STEPHEN J. FINK. "Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers)". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 1–12. ISBN: 978-1-4503-1205-9. DOI: http://dx.doi.org/10.1145/2254064.2254066.

[Dyb17] DYBDAL, MARTIN. "Array abstractions for GPU programming". PhD thesis. Department of Computer Science, Faculty of Science, University of Copenhagen, 2017.

[ED14] ELSMAN, MARTIN and MARTIN DYBDAL. "Compiling a Subset of APL Into a Typed Intermediate Language". In: *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[FD02] FAHNDRICH, MANUEL and ROBERT DELINE. "Adoption and Focus: Practical Linear Types for Imperative Programming". In: *SIGPLAN Not.* 37.5 (May 2002), pp. 13–24. ISSN: 0362-1340. DOI: http://dx.doi.org/10.1145/543552.512532. URL: http://doi.acm.org/10.1145/543552.512532.

[Fra04] FRASER, KEIR. *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[GHS06] GRELCK, CLEMENS, KARSTEN HINCKFUSS, and SVEN-BODO SCHOLZ. "With-Loop Fusion for Data Locality and Parallelism". In: *Proceedings of the 17th International Conference on Implementation and Application of Functional Languages*. IFL'05. Dublin, Ireland: Springer-Verlag, 2006, pp. 178–195. DOI: http://dx.doi.org/10.1007/11964681_11. URL: http://dx.doi.org/10.1007/11964681_11.

[Gro+14] GROSSER, TOBIAS, ALBERT COHEN, JUSTIN HOLEWINSKI, P. SADAYAPPAN, and SVEN VERDOOLAEGE. "Hybrid Hexagonal/Classical Tiling for GPUs". In: *Procs. Int. Symposium on Code Generation and Optimization*. CGO '14. ACM, 2014, 66:66–66:75. DOI: http://dx.doi.org/10.1145/2544137.2544160.

[GS06] GRELCK, CLEMENS and SVEN-BODO SCHOLZ. "SAC - A Functional Array Language for Efficient Multi-Threaded Execution". In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427.

[GTA06] GORDON, MICHAEL I., WILLIAM THIES, and SAMAN AMARASINGHE. "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs". In: *Procs. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 151–162. ISBN: 1-59593-451-0. DOI: http://dx.doi.org/10.1145/1168857.1168877. URL: http://doi.acm.org/10.1145/1168857.1168877.

[Hen+16] HENRIKSEN, TROELS, MARTIN DYBDAL, HENRIK URMS, ANNA SOFIE KIEHN, DANIEL GAVIN, HJALTE ABELSKOV, MARTIN ELSMAN, and COSMIN OANCEA. "APL on GPUs: A TAIL from the Past, Scribbled in Futhark". In: *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*. FHPC'16. Nara, Japan: ACM, 2016, pp. 38–43.

[Hen+17] HENRIKSEN, TROELS, NIELS GW SERUP, MARTIN ELSMAN, FRITZ HENGLEIN, and COSMIN 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*. ACM. 2017, pp. 556–571.

[Hen14] HENRIKSEN, TROELS. "Exploiting functional invariants to optimise parallelism: a dataflow approach". In: *Master's thesis, DIKU, Denmark* (2014).

[HEO14] HENRIKSEN, TROELS, MARTIN ELSMAN, and COSMIN E OANCEA. "Size slicing: a hybrid approach to size inference in Futhark". In: *Proc. of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*. ACM. 2014, pp. 31–42.

[HG92] HENDREN, LAURIE J and GUANG R GAO. "Designing programming languages for analyzability: A fresh look at pointer data structures". In: *Computer Languages, 1992., Proceedings of the 1992 International Conference on*. IEEE. 1992, pp. 242–251.

[Hil89] HILLIS, W DANIEL. *The connection machine*. MIT press, 1989.

[HLO16] HENRIKSEN, TROELS, KEN FRIIS LARSEN, and COSMIN 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*. ARRAY 2016. Santa Barbara, CA, USA: ACM, 2016, pp. 17–24.

[HM93] HAINS, G. and L. M. R. MULLIN. "Parallel Functional Programming with Arrays". In: *The Computer Journal* 36.3 (1993), p. 238. DOI: http://dx.doi.org/10.1093/comjnl/36.3.238. eprint: /oup/backfile/content_public/journal/comjnl/36/3/10.1093/comjnl/36.3.238/2/360238.pdf. URL: +%20http://dx.doi.org/10.1093/comjnl/36.3.238.

[HO13] HENRIKSEN, TROELS and COSMIN EUGEN OANCEA. "A T2 graph-reduction approach to fusion". In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM. 2013, pp. 47–58.

[HO14] HENRIKSEN, TROELS and COSMIN E OANCEA. "Bounds checking: An instance of hybrid analysis". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM. 2014, p. 88.

[Hoa13] HOARE, GRAYDON. *The Rust Programming Language*. June 2013. URL: http://www.rust-lang.org/.

[Hol+14] HOLK, ERIC, RYAN NEWTON, JEREMY SIEK, and ANDREW LUMSDAINE. "Region-based memory management for GPU programming languages: enabling rich data structures on a spartan host". In: *ACM SIGPLAN Notices* 49.10 (2014), pp. 141–155.

[Hor+11] HORMATI, AMIR H., MEHRZAD SAMADI, MARK WOH, TREVOR MUDGE, and SCOTT MAHLKE. "Sponge: Portable Stream Programming on Graphics Engines". In: *Procs. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 381–392. ISBN: 978-1-4503-0266-1. DOI: http://dx.doi.org/10.1145/1950365.1950409. URL: http://doi.acm.org/10.1145/1950365.1950409.

[Ive62] IVERSON, KENNETH E. *A Programming Language*. John Wiley and Sons, Inc, 1962.

[Jay99] JAY, C. BARRY. "Programming in FISh". In: *International Journal on Software Tools for Technology Transfer* 2.3 (1999), pp. 307–315.

[Jon92] JONES, SIMON L PEYTON. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". In: *Journal of functional programming* 2.2 (1992), pp. 127–202.

[JTH01] JONES, SIMON PEYTON, ANDREW TOLMACH, and TONY HOARE. "Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC". In: *Haskell Workshop*. Vol. 1. 2001, pp. 203–233.

[KA02] KENNEDY, KEN and JOHN R. ALLEN. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0.

[Kel+10] KELLER, GABRIELE, MANUEL MT CHAKRAVARTY, ROMAN LESHCHINSKIY, SIMON PEYTON JONES, and BEN LIPPMEIER. "Regular, Shape-Polymorphic, Parallel Arrays in Haskell". In: *ACM Sigplan Notices* 45.9 (2010), pp. 261–272.

[Kel+12] KELLER, GABRIELE, MANUEL M.T. CHAKRAVARTY, ROMAN LESHCHINSKIY, BEN LIPPMEIER, and SIMON PEYTON JONES. "Vectorisation Avoidance". In: *Proceedings of the 2012 Haskell Symposium*. Haskell '12. Copenhagen, Denmark: ACM, 2012, pp. 37–48. ISBN: 978-1-4503-1574-6. DOI: http://dx.doi.org/10.1145/2364506.2364512. URL: http://doi.acm.org/10.1145/2364506.2364512.

[Klö+12] KLÖCKNER, ANDREAS, NICOLAS PINTO, YUNSUP LEE, B. CATANZARO, PAUL IVANOV, and AHMED FASIH. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2011.09.001.

[KM93] KENNEDY, KEN and KATHRYN S MCKINLEY. "Maximizing loop parallelism and improving data locality via loop fusion and distribution". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 1993, pp. 301–320.

[Kri+16] KRISTENSEN, MADS R.B., SIMON A.F. LUND, TROELS BLUM, and JAMES AVERY. "Fusion of Parallel Array Operations". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel: ACM, 2016, pp. 71–85. ISBN: 978-1-4503-4121-9. DOI: http://dx.doi.org/10.1145/2967938.2967945. URL: http://doi.acm.org/10.1145/2967938.2967945.

[Lee+14] LEE, HYOUKJOONG, KEVIN J. BROWN, ARVIND K. SUJEETH, TIARK ROMPF, and KUNLE OLUKOTUN. "Locality-Aware Mapping of Nested Parallel Patterns on GPUs". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 63–74. ISBN: 978-1-4799-6998-2. DOI: http://dx.doi.org/10.1109/MICRO.2014.23.

[Les09] LESHCHINSKIY, ROMAN. "Recycle Your Arrays!" In: *Practical Aspects of Declarative Languages: 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings*. Ed. by GILL, ANDY and TERRANCE SWIFT. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 209–223. ISBN: 978-3-540-92995-6. DOI: http://dx.doi.org/10.1007/978-3-540-92995-6_15. URL: https://doi.org/10.1007/978-3-540-92995-6_15.

[LH17] LARSEN, RASMUS WRIEDT and TROELS HENRIKSEN. "Strategies for Regular Segmented Reductions on GPU". In: *Proceedings of the 6th ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '17. New York, NY, USA: ACM, 2017.

[McD+13] MCDONELL, TREVOR L., MANUEL MT CHAKRAVARTY, GABRIELE KELLER, and BEN LIPPMEIER. "Optimising Purely Functional GPU Programs". In: *Procs. of Int. Conf. Funct. Prog. (ICFP)*. 2013.

[MF16] MADSEN, FREDERIK M and ANDRZEJ FILINSKI. "Streaming nested data parallelism on multicores". In: *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM. 2016, pp. 44–51.

[MFP91] MEIJER, ERIK, MAARTEN FOKKINGA, and ROSS PATERSON. "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire". In: *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA)*. Cambridge, MA, 1991, pp. 26–30.

[MP90] MIDKI, SAMUEL P and DAVID A PADUA. "Issues in the Compile-Time Optimization of Parallel Programs". In: *Procs. of Int. Conf. on Parallel Processing*. Vol. 2. 1990, pp. 105–113.

[MS97] MEGIDDO, NIMROD and VIVEK SARKAR. "Optimal Weighted Loop Fusion for Parallel Programs". In: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '97. Newport, Rhode Island, USA: ACM, 1997, pp. 282–291. ISBN: 0-89791-890-8. DOI: http://dx.doi.org/10.1145/258492.258520. URL: http://doi.acm.org/10.1145/258492.258520.

[MTM97] MILNER, ROBIN, MADS TOFTE, and DAVID MACQUEEN. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.

[MYB16] MALEKI, SEPIDEH, ANNIE YANG, and MARTIN BURTSCHER. "Higher-order and Tuple-based Massively-parallel Prefix Sums". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, 2016, pp. 539–552. ISBN: 978-1-4503-4261-2. DOI: http://dx.doi.org/10.1145/2908080.2908089. URL: http://doi.acm.org/10.1145/2908080.2908089.

[Oan+12] OANCEA, COSMIN, CHRISTIAN ANDREETTA, JOST BERTHOLD, ALAIN FRISCH, and FRITZ HENGLEIN. "Financial Software on GPUs: between Haskell and Fortran". In: *Funct. High-Perf. Comp. (FHPC'12)*. 2012.

[OM08] OANCEA, COSMIN E. and ALAN MYCROFT. "Software Thread-Level Speculation – An Optimistic Library Implementation". In: *IWMSE*. 2008.

[OR11] OANCEA, COSMIN E. and LAWRENCE RAUCHWERGER. "A Hybrid Approach to Proving Memory Reference Monotonicity". In: *Procs. Int. Lang. Comp. Par. Comp. (LCPC)*. 2011.

[OR12] OANCEA, COSMIN E. and LAWRENCE RAUCHWERGER. "Logical Inference Techniques for Loop Parallelization". In: *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*. 2012, pp. 509–520.

[OR15] OANCEA, COSMIN E. and LAWRENCE RAUCHWERGER. "Scalable Conditional Induction Variables (CIV) Analysis". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '15. San Francisco, California: IEEE Computer Society, 2015, pp. 213–224. ISBN: 978-1-4799-8161-8. URL: http://dl.acm.org/citation.cfm?id=2738600.2738627.

[OW05] OANCEA, C. E. and S. M. WATT. "Domains and Expressions: An Interface between Two Approaches to Computer Algebra". In: *Proceedings of the ACM ISSAC 2005*. 2005, pp. 261–269. URL: http://www.csd.uwo.ca/~coancea/Publications.

[PM15] PRICE, JAMES and SIMON MCINTOSH-SMITH. "Oclgrind: An extensible OpenCL device simulator". In: *Procs. of the 3rd Int. Workshop on OpenCL*. ACM. 2015, p. 12.

[Pou+11] POUCHET, LOUIS-NOËL, UDAY BONDHUGULA, CÉDRIC BASTOUL, ALBERT COHEN, J. RAMANUJAM, P. SADAYAPPAN, and NICOLAS VASILACHE. "Loop Transformations: Convexity, Pruning and Optimization". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 549–562. ISBN: 978-1-4503-0490-0. DOI: http://dx.doi.org/10.1145/1926385.1926449. URL: http://doi.acm.org/10.1145/1926385.1926449.

[PPS96] PEYTON JONES, SIMON, WILL PARTAIN, and ANDRÉ SANTOS. "Let-floating: Moving Bindings to Give Faster Programs". In: *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*. ICFP '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 1–12. ISBN: 0-89791-770-7. DOI: http://dx.doi.org/10.1145/232627.232630. URL: http://doi.acm.org/10.1145/232627.232630.

[Rag+13] RAGAN-KELLEY, JONATHAN, CONNELLY BARNES, ANDREW ADAMS, SYLVAIN PARIS, FRÉDO DURAND, and SAMAN AMARASINGHE. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: http://dx.doi.org/10.1145/2491956.2462176.

[Rey72] REYNOLDS, JOHN C. "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM annual conference-Volume 2*. ACM. 1972, pp. 717–740.

[Ric53] RICE, H. G. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 00029947. URL: http://www.jstor.org/stable/1990888.

[RKC16] REDDY, CHANDAN, MICHAEL KRUSE, and ALBERT COHEN. "Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel: ACM, 2016, pp. 87–97. ISBN: 978-1-4503-4121-9. DOI: http://dx.doi.org/10.1145/2967938.2967950. URL: http://doi.acm.org/10.1145/2967938.2967950.

[RLK14] ROBINSON, AMOS, BEN LIPPMEIER, and GABRIELE KELLER. "Fusing Filters with Integer Linear Programming". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '14. Gothenburg, Sweden: ACM, 2014, pp. 53–62. ISBN: 978-1-4503-3040-4. DOI: http://dx.doi.org/10.1145/2636228.2636235. URL: http://doi.acm.org/10.1145/2636228.2636235.

[SF92] SABRY, AMR and MATTHIAS FELLEISEN. "Reasoning About Programs in Continuation-passing Style." In: *SIGPLAN Lisp Pointers* V.1 (Jan. 1992), pp. 288–298. ISSN: 1045-3563.

[SGS10] STONE, JOHN E., DAVID GOHARA, and GUOCHUN SHI. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. ISSN: 0740-7475. DOI: http://dx.doi.org/10.1109/MCSE.2010.69. URL: http://dx.doi.org/10.1109/MCSE.2010.69.

[Sha+17] SHAIKHHA, AMIR, ANDREW FITZGIBBON, SIMON PEYTON-JONES, and DIMITRIOS VYTINIOTIS. "Destination-Passing Style for Efficient Memory Management". In: FHPC '17 (2017).

[Spo+08] SPOONHOWER, DANIEL, GUY E. BLELLOCH, ROBERT HARPER, and PHILLIP B. GIBBONS. "Space Profiling for Parallel Functional Programs". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: ACM, 2008, pp. 253–264. ISBN: 978-1-59593-919-7. DOI: http://dx.doi.org/10.1145/1411204.1411240. URL: http://doi.acm.org/10.1145/1411204.1411240.

[SRD17] STEUWER, MICHEL, TOOMAS REMMELG, and CHRISTOPHE DUBACH. "Lift: A Functional Data-parallel IR for High-performance GPU Code Generation". In: *Procs. of Int. Symp. on Code Generation and Optimization*. CGO'17. Austin, USA: IEEE Press, 2017, pp. 74–85. ISBN: 978-1-5090-4931-8. URL: http://dl.acm.org/citation.cfm?id=3049832.3049841.

[Ste+15] STEUWER, MICHEL, CHRISTIAN FENSCH, SAM LINDLEY, and CHRISTOPHE DUBACH. "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code". In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 205–217. ISSN: 0362-1340. DOI: http://dx.doi.org/10.1145/2858949.2784754. URL: http://doi.acm.org/10.1145/2858949.2784754.

[Str+12] STRATTON, JOHN A, CHRISTOPHER RODRIGUES, I-JUI SUNG, NADY OBEID, LI-WEN CHANG, NASSER ANSSARI, GENG DANIEL LIU, and WEN-MEI W HWU. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* 127 (2012).

[TC13] THIEMANN, PETER and MANUEL M. T. CHAKRAVARTY. "Agda Meets Accelerate". In: *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*. IFL'2012. Revised Papers, Springer-Verlag, LNCS 8241. 2013.

[TG09] TROJAHNER, KAI and CLEMENS GRELCK. "Dependently typed array programs don't go wrong". In: *The Journal of Logic and Algebraic Programming* 78.7 (2009). The 19th Nordic Workshop on Programming Theory (NWPT'2007), pp. 643–664.

[TG11] TROJAHNER, KAI and CLEMENS GRELCK. "Descriptor-free Representation of Arrays with Dependent Types". In: *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*. IFL'08. Hatfield, UK: Springer-Verlag, 2011, pp. 100–117.

[Tof+04] TOFTE, MADS, LARS BIRKEDAL, MARTIN ELSMAN, and NIELS HALLENBERG. "A Retrospective on Region-Based Memory Management". In: *Higher-Order and Symbolic Computation (HOSC)* 17.3 (Sept. 2004), pp. 245–265.

[TP11] TOV, JESSE A. and RICCARDO PUCELLA. "Practical Affine Types". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 447–458. ISBN: 978-1-4503-0490-0. DOI: http://dx.doi.org/10.1145/1926385.1926436. URL: http://doi.acm.org/10.1145/1926385.1926436.

[TPO06] TARDITI, DAVID, SIDD PURI, and JOSE OGLESBY. *Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses*. Tech. rep. Oct. 2006, p. 11. URL: https://www.microsoft.com/en-us/research/publication/accelerator-using-data-parallelism-to-program-gpus-for-general-purpose-uses/.

[Vej94] VEJLSTRUP, MAGNUS. "Multiplicity Inference". MA thesis. Department of Computer Science, University of Copenhagen, Sept. 1994.

[Ver+13] VERDOOLAEGE, SVEN, JUAN CARLOS JUEGA, ALBERT COHEN, JOSÉ IGNACIO GÓMEZ, CHRISTIAN TENLLADO, and FRANCKY CATTHOOR. "Polyhedral Parallel Code Generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 54:1–54:23. ISSN: 1544-3566. DOI: http://dx.doi.org/10.1145/2400682.2400713. URL: http://doi.acm.org/10.1145/2400682.2400713.

[Wat+90] WATT, S. M., R. D. JENKS, R. S. SUTOR, and B. M. TRAGER. "The Scratchpad II Type System: Domains and Subdomains". In: *Procs of Computing Tools For Scientific Problem Solving*. A. Miola ed. Academic Press, 1990, pp. 63–82.

[Wat03] WATT, S. M. "Aldor". In: *Handbook of Computer Algebra*. Ed. by GRABMEIER, J., E. KALTOFEN, and V. WEISPFENNING. 2003, pp. 154–160.

[WM95] WULF, WM. A. and SALLY A. MCKEE. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: http://dx.doi.org/10.1145/216585.216588. URL: http://doi.acm.org/10.1145/216585.216588.

[WY14] WANG, JIN and SUDHAKAR YALAMANCHILI. "Characterization and analysis of dynamic parallelism in unstructured GPU applications". In: *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE. 2014, pp. 51–60.

[Yan+10] YANG, YI, PING XIANG, JINGFEI KONG, and HUIYANG ZHOU. "A GPGPU Compiler for Memory Optimization and Parallelism Management". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, pp. 86–97. ISBN: 978-1-4503-0019-3. DOI: http://dx.doi.org/10.1145/1806596.1806606. URL: http://doi.acm.org/10.1145/1806596.1806606.