UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Static and Dynamic Analyses for Efficient GPU Execution

Philip Munksgaard

Advisors: Cosmin Oancea and Troels Henriksen

Handed in: April 11, 2023

This thesis has been submitted to the PhD School of The Faculty of Science, University of Copenhagen

Abstract

In this thesis we describe a host of static and dynamic techniques for efficient execution of GPU programs.

Most significant is the *array short-circuiting* technique, which automatically rewrites array updates and concatenations to happen in-place when deemed safe. The optimization is based on FunMem, an intermediate representation with non-semantic memory information that we also introduce. FunMem allows the compiler to analyze, reason about and optimize memory usage patterns while keeping high-level language information. To map array elements to specific locations in memory, FunMem uses LMADs. LMADs have traditionally seen most use in the context of automatic parallelization of sequential loops, where they are used in a set-interpretation to aggregate memory accesses across loops, but we introduce two new interpretations: as index functions in FunMem and as a slicing mechanism in a user-facing language. The first enable cheap change-of-layout transformations in FunMem, while the other allows users to express complex slices of arrays not otherwise possible. Finally, to enable the aforementioned short-circuiting optimization on complex benchmarks, we present a heuristic for proving non-overlap of multi-dimensional LMADs in the set-interpretation.

We also introduce a technique for automatically finding near-optimal threshold values for multi-versioned code. Multi-versioned code attempts to solve the problem of not having a single best compilation strategy for a certain program, by generating multiple semantically-equivalent but differently-optimized versions of code and guarding each version with a conditional based on runtime parameters, leading to the creation of a tuning tree. By relying on a *monotonicity assumption*, which relates the relative performance of each code version to the runtime parameter conditionals, we show how to automatically tune the threshold values such that each tuning dataset is executed using the fastest code version. As a result, our one-time offline tuning process produces tuning values that optimally discriminate between the different code versions for the training datasets used, and indeed for all datasets with similar size-characteristics.

Both the static and dynamic techniques have been developed in the context of and implemented for the Futhark parallel programming language, and we show significant performance improvements from both array short-circuiting and autotuning.

Resumé

I denne afhandling beskriver vi en række statiske og dynamiske teknikker målrettet effektiv afvikling af GPU programmer.

Den mest signifikante af disse er *geledskortslutningsteknikken*, som automatisk omskriver geledopdateringer og sammensætninger til at foretage direkte ændringer i geledder når det er sikkert. Optimeringen er baseret på FunMem, en mellemliggende repræsentation med ikke-semantisk hukommelsesinformation, som vi også introducerer. FunMem lader oversætteren analysere og optimere hukommelsesforbrug og -mønstre, samtidig med at høj-niveau sprog-information beholdes. For at binde geledelementer til placeringer i hukommelsen bruges LMADs. LMADs er traditionelt blevet brugt i forbindelse med automatisk parallelisering af sekventielle løkker, hvor de i en sætfortolkning samler information om hukommelsestilgange. Vi introducerer to nye fortolkninger: Som indeksfunktioner i FunMem og som en udsnitsmekanisme i et brugervendt sprog. Den første fortolkning lader FunMem udtrykke billige indretningsændringsomdannelser, mens den anden lader brugeren udtrykke komplicerede geledudsnit som ikke ellers er mulige. For at kunne bruge førnævnte geledskortslutningsteknik til komplicerede programmer beskriver vi også en metode til at bestemme om two flerdimensionelle LMADs i overlapper eller ej.

Derudover vises en teknik til automatisk at finde nær-optimale tærskelværdier i flerversioneret kode. Flerversioneret kode bruges til at løse problemet med at der for nogle programmer ikke findes én bedste oversættelsesstrategi. Ved at lave flere semantisk ens men forskelligt optimerede versioner af den samme kode kan der dynamisk vælges mellem dem på køretidspunktet. Hver version beskyttes af en køretidssammenligning mellem en brugerdefineret tærskelværdi og en dynamisk programværdi, hvilket tilsammen bliver til et stemmetræ. Ved at forlade os på en ensformighedsantagelse som relaterer den relative ydelse af hver kodeversion til de dynamiske programværdier, viser vi hvordan tærskelværdierne kan stemmes således at alle stemmedatasæt afvikles med hver deres hurtigste kodeversion. Som resultat kan vi ved blot at stemme træet en gang opnå ét sæt tærskelværdier der automatisk vælger den bedste kodeversion for alle stemmedatasæt, samt alle andre datasæt med lignende størrelseskarakteristika.

Alle statiske og dynamiske teknikker i denne afhandling er blevet implementeret til det parallelle programmeringssprog Futhark, og vi finder at især geledkortslutningsteknikken og den automatiske tærskelværdistemmer giver væsentlige ydelsesforbedringer.

Contents

List of Figures

List of Tables

List of Procedures

List of Listings

Dedication

To Ebbe.

Acknowledgements

The research presented in this thesis has been partially supported by the Independent Research Fund Denmark grant under the research project *FUTHARK: Functional Technology for High-performance Architectures*.

The illustration on the cover was created with the assistance of DALL·E 2, a deep learning model created by OpenAI. Modern deep learning techniques makes heavy use of GPUs. Hedgehogs are the official mascot of the Futhark programming language.

First of all, I want to thank my co-authors on the various papers I have worked on during the last three years: Svend Lund Breddam, Fabian Cristian Gieseke, Ponnuswamy Sadayappan, Troels Henriksen and Cosmin Oancea. It has been a joy working with you.

Second of all, none of this would have happened without my supervisors, Cosmin Oancea and Troels Henriksen. I am extremely grateful that you have taken me under your collective wings.

Finally, I am thankful for the support I have received from my family, especially Ida, who has been with me on every step of this journey.

Chapter 1

Introduction

This thesis describes techniques for automatically performing analyses and optimizations in GPU-oriented programming languages. It consists of two distinct but related lines of work:

1. Static analysis tools and techniques for reasoning about and optimizing memory usage in languages targeting GPU execution.

2. Dynamic analyses aimed at automatically tuning threshold parameters in multi-versioned code based on monotonic properties.

Common to both approaches is the context: Attempting to automatically optimize the performance of GPU oriented programs.

1.1 Static Memory Analyses in Parallel Languages

Efficient use of memory is crucial to the performance of many programs, especially programs targeted at GPU execution [Yan+10]. Modern GPUs have two main levels of memory, which we denote *global* and *local* memory [1], which must be managed judiciously by the user. Correctly using *coalesced access patterns* and avoiding *bank conflicts* can lead to performance improvements of up to an order of magnitude. The imperative languages commonly used to program GPUs, CUDA and OpenCL, require the programmer to directly manage these concerns by manually allocating memory in different parts of the memory hierarchy and mapping the results to logical arrays, which becomes increasingly difficult as programs grow in complexity. Similarly, flattened indices and the implicit structure of parallelism makes the resulting code difficult to later maintain, reason about and optimize.

A rich amount of work concerns automatic parallelization of sequential code such as the work on the SUIF [Hal+05; MH99] and Polaris [RHR03; OR12] compilers as well as work that uses polyhedral compilation [Bon+08; CSS15].

[1] Local memory is called *shared* memory in CUDA terminology.

These analyses are complicated by their attempt to reverse-engineer the users' memory optimizations. Inferring the parallel structure of a given program might fail, in which case the program is executed without any parallelism at all, leading to catastrophic performance.

Parallel functional array languages [Hen+19; Mac+19; Hal+05] represent an alternative approach, where all available parallelism is explicitly denoted by the programmer using second-order array combinators such as `map`, `reduce` and `scan`, leading to correct-by-construction parallelism. While promising, this approach separates the notion of memory from the notion of arrays, removing the users' ability to reason about and optimize memory usage at all. As a result, many common optimizations from imperative languages, such as in-place updates or reusing already allocated memory are inexpressible, leading to unnecessarily high memory usage.

As a solution, we present a series of analyses, tools and optimizations for reasoning about and optimizing the use of memory in parallel languages, all based on the use of LMADs [PHP98; Hoe98].

LMADs, and various extensions thereof, have traditionally been used to perform automatic parallelization by aggregating array accesses across sequential loops and performing complex set-operations on the result [RHR03; OR13]. We will also be using LMADs in this capacity but we describe a new heuristic for proving *non-overlap of multi-dimensional* LMADs. In addition to this interpretation, we introduce LMADs as a *slicing construct* in the user-facing language, allowing users to express complex array "views" not otherwise possible, and as *index functions* in the intermediate representation of an optimizing compiler, enabling free change-of-layout transformations of arrays.

We next introduce an intermediate representation based on LMADs, FunMem, that can be used by the compiler for a parallel array language to introduce non-semantic memory information without lowering the program to an imperative setting. By associating with each array an index function consisting of an LMAD, the compiler can easily manipulate and optimize memory usage and patterns, for instance to ensure coalesced access to local memory, allow safe in-place updates of arrays or enable memory reuse, without changing the semantics of the program. We formalize our discussion of FunMem by showing a translation from a pure functional IR named Fun and a translation to an lower-level imperative language named Imp, as well as static and dynamic semantics for each language.

Finally, we describe three optimizations based on FunMem that have all been implemented in the Futhark programming language: Memory expansion, memory block merging and array short-circuiting.

Memory expansion arises from the fact that GPUs do not efficiently support dynamic allocation of memory inside threads. Therefore, all allocations of arrays have to be hoisted out of the kernels themselves and expanded to accommodate all threads in one allocation. We show how memory expansion is trivially implemented using the FunMem IR.

Memory block merging aims to reduce memory usage by applying register allocation techniques on arrays. By computing an interference graph between the memory blocks of a given program we are able to take into account the different memory spaces and additional requirements imposed by the GPU context, while merging memory blocks and reducing memory usage. We have implemented memory block merging in the Futhark compiler, and though it applies to many of the programs in the Futhark benchmark suite, it turns out to have limited impact in the already highly optimized compiler.

Array short-circuiting is a complex analysis that enables compilers to recapture some of the lost memory optimizations arising from parallelism guarantees in high-level parallel programming languages. These languages enforce correct-by-construction parallelism using traditional type system features by forcing a separation between parallel operations that read and write to the same array. An in-place update of the elements of an array is therefore always divided into two separate parallel operations: First computing the updated values in a intermediate array that is manifested to memory and secondly updating the original array using the values from the intermediate array. The result is extra overhead from copying and an increased memory footprint. Array short-circuiting consists of a bottom-up analysis that is based on aggregating array accesses using LMADs and proving disjointedness of reads and writes. The aim is to identify cases where it is safe to construct the intermediate array directly in the memory space of the original array, such that the parallel write becomes a no-op. The result is that in-place updates reading and writing to the same array can be constructed without any intermediate arrays when deemed safe. We have implemented this optimization in the Futhark compiler and report a performance speedup on six public benchmarks of between 1.1× and 2×, resulting in code that is competitive with hand-written OpenCL and CUDA code.

1.2 Autotuning Threshold Parameters in Multi-Versioned Code

For many applications, it can be difficult to statically infer the best compilation technique [Che+; Fur+11; Aca+19; TJF14]. In fact, for any given program, there may not exist *one* compilation technique that results in optimal performance across all datasets and hardware specifications [Ste+; Bag+15; Rag+13; Fra+18; Hag+18; Aca+19; TJF14]. This is especially true in the context of massively-parallel hardware such as GPUs, where the performance of a given program is extremely sensitive to locality, memory usage, parallelization strategies, differences in hardware characteristics as well as dataset sizes and shapes [Jan+10; Hij+22].

A promising strategy to deal with this problem is to generate *multi-versioned code* consisting of many semantically-equivalent but differently-optimized ver-

sions of the same computational kernel and then determining at runtime which version to run by comparing dynamic runtime properties (specifically, the degree of parallelism exploited) to user-defined threshold values [Hen+19]. The different code versions and the conditionals discriminating between them form a *tuning tree*. In theory, by adjusting the threshold values of the tuning tree, users can target the best code-version for a given dataset and hardware combination. In practice, that is not feasible for most users, given the complicated nature of the tuning trees and the intricate knowledge of the GPU architecture required.

Instead we propose an automatic tuning technique which is based on the structure of the tuning tree and relies on a *monotonicity assumption*, which relates the relative performance of each code version to the degrees of parallelism exploited. Arising from compiler-autotuner co-design it requires minimal compiler instrumentation and can be applied to any tuning tree that conforms to the monotonicity assumption. When the monotonicity assumption is upheld, the proposed autotuner guarantees *near-optimal* discrimination of code versions for all datasets which have similar parallelism characteristics to the datasets being used to tune with.

For a given dataset, our autotuner relies on finding for each tuning parameter the *maximal optimal threshold interval*, which causes the dataset to be executed using the best code version. The monotonicity assumption then guarantees that for each such interval found, we can intersect it with the intervals found for all other datasets. The result is a set of threshold intervals that optimally select between the different code versions for all the given tuning datasets, as well as any other datasets with similar parallelism characteristics. In addition to describing this process in detail, we also outline a proof for why adherence to the monotonicity assumption guarantees that such intersections exist.

Furthermore, we describe two classes of programs, *size-invariant* and *size-variant* programs, and argue for why tuning size-variant programs can be reduced to instances of tuning size-invariant programs. We also show how the monotonicity assumption gives rise to a binary search technique that can be used to reduce the amount of tuning runs necessary to tune size-variant programs.

Lastly, we demonstrate the usefulness of our autotuning technique by implementing it for the Futhark programming language and comparing the tuning time and resulting program performance to that of an OpenTuner-based autotuner previously developed for Futhark, using a variety of public benchmarks. We find that our autotuner reduces the tuning time up to 22.6× and finds better threshold values in five out of 11 cases, leading to an increased benchmark performance of up to 10×.

Chapter 2

Background

This chapter contains context and background information necessary for understanding the work presented in the following chapters. The work that we present is based on the Futhark programming language [Hen+19; Hen+17; Hen17], a pure functional array language targeted at GPU execution (though other backends are also available), so we need to understand both how GPUs work and how Futhark maps computations to GPU kernels.

The purpose of this chapter is not to give a comprehensive and detailed description of either GPUs or Futhark and their inner workings, but to give a cursory introduction which will serve as a suitable base for understanding the work presented in the rest of this thesis. For additional details about GPUs we refer to the official CUDA documentation at `https://docs.nvidia.com/cuda` and the official OpenCL site at `https://www.khronos.org/opencl/`. For more information about Futhark, we refer to `https://futhark-lang.org/`.

2.1 Parallelism and GPUs

Graphics processing units, or GPUs for short, were popularized in the '90s and early '00s for handling the increasingly complex 3D graphics in video games. They have since become popular platforms for high-performance computing and data-processing under the term GPGPU (General Purpose GPU), though we will use the shorter GPU.

The distinguishing feature of GPUs are their single-instruction multiple-thread (SIMT) architecture, which allows many threads to run in parallel. For example, NVIDIA's A100 GPU [Nvi22], which is the one we'll be using for most of the benchmarks in this thesis, has 6912 cores, meaning that 6912 threads can run at the same time. These 6912 cores are divided over 108 *streaming multiprocessors* each with 64 cores that can process in parallel. Furthermore, modern GPUs make aggressive use of multi-threading to hide latency, so the number of hardware threads needed to reach full utilization and saturation

is in the hundreds of thousands. All in all, GPUs have massive amounts of parallelism available to them.

At the software level, the parallelism of a GPU like the A100 is defined in three levels: Grid, workgroup[1] and warp, with the last one not being explicitly addressable. A grid is the main unit of execution and it consists of a number of workgroups, arranged in one, two or three dimensions, though we will only focus on the one-dimensional case.

A workgroup consists of a number of threads, up to 1024 on most modern GPUs which are again divided into warps, normally consisting of 32 threads. Though each thread has its own program counter and register state, the threads of a warp always execute one common instruction at a time. This means that branching can cause some threads to be idle while others are working, but if there is at least one thread in each branch of a conditional, the entire warp needs to execute all branches, one after the other. As a result, full efficiency is only achieved when all 32 threads in a warp are executing the same instructions.

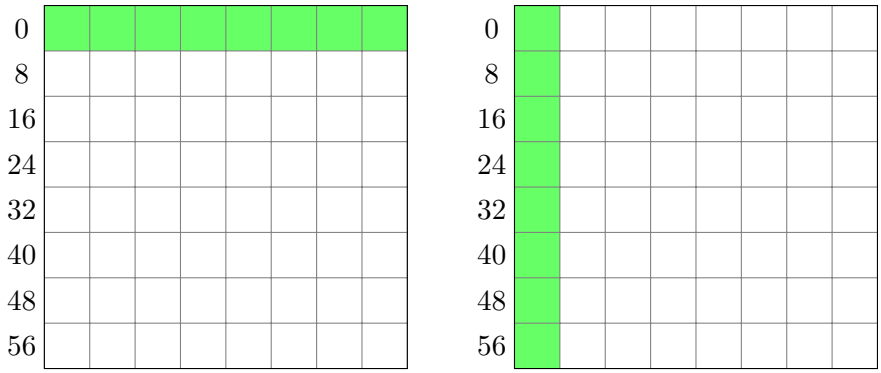
Inter-thread communication is achieved through the use of different kinds of memory. At the grid level we have global memory, the slowest and largest kind but also the only one that can be used for inter-group communication. The A100 has either 40 or 80 GB of global memory. Threads in the same workgroup may use local memory[2] to communicate between themselves. Local memory is much faster than global memory, but also much smaller: Each workgroup has a maximum of 48 KB (which can be reconfigured up to 164 KB on the A100) of local memory available. Thus, a common pattern is to attempt to chunk a given problem into blocks that fit in the local memory of a workgroup such that there is some kind of cooperation between the workgroup threads, in order to take advantage of the fast local memory. Finally, threads in a workgroup can synchronize their execution through the use of barriers, but there is no way to synchronize across workgroups.

When transferring data from global memory to local memory or vice versa, the speed of memory accesses can be significantly improved by using *coalesced access*. Put simply, if the threads of a warp all attempt to access global memory addresses that are consecutive in memory, they are able to do so in a single memory operation. In contrast, if the locations accessed are far apart in memory the warp will have to issue distinct memory operations for each thread. Figure 2.1 shows examples of coalesced and uncoalesced accesses to memory.

Additionally, while working with local memory, the programmer has to be aware of *bank conflicts*. Local memory is divided into banks (the A100 has 32 banks), such that successive words in memory belong to successive banks. As a result, two memory locations separated by e.g. 32 words belong to the same bank. Any number of banks can be used in parallel, but if two or more threads

[1] In CUDA terminology, these are called blocks, but we will be using the term workgroup from OpenCL terminology.

[2] Called shared memory in CUDA terminology.



(a) Coalesced accesses to memory. **(b)** Uncoalesced accesses to memory.

Figure 2.1: Coalesced and uncoalesced memory access patterns. The green locations are being accessed by consecutive threads in the same warp at the same time.

```
kernel plus_one(in, out) :
  tid ← get_local_id()
  wid ← get_workgroup_id()
  gid ← wid * WORKGROUP_SIZE + tid
  out[gid] ← in[gid] + 1
```

Listing 2.1: Computing the incremented values of *in* in parallel.

in the same warp concurrently attempt to access elements in the same bank then those accesses will be sequentialized.

GPU code is organized into *kernels*. A kernel is a function that takes a number of buffers as input and is executed on a single thread. Each thread has a local ID, unique to the workgroup it is in, a workgroup ID and a global ID unique across all threads. These IDs can be accessed by using the *get_local_id()* and *get_workgroup_id()* functions. For example, listing 2.1 shows the pseudo-code for a kernel that takes as arguments an input buffer *in* and an output buffer *out*, increments all the values of *in* by one and writes the result to the output buffer *out*.

Due to the physical constraints of the GPU, kernels are quite limited in what they can do. For instance, neither recursion nor dynamic memory allocation are supported (at least not efficiently). Therefore, all memory used by the kernels must be pre-allocated by the host and passed as arguments.

2.2 Futhark

Futhark is a pure functional parallel array programming language. The primary target of the language is GPU execution, which is also the focus of this thesis.

```
map ‘a [n] ‘x : (f : a → x) → (as : [n]a) → *[n]x
reduce [n] ‘a : (op : a → a → a) → (ne : a) → (as : [n]a) → a
scan [n] ‘a : (op : a → a → a) → (ne : a) → (as : [n]a) → *[n]a
```

Listing 2.2: Common SOACs and their types.

Futhark also has support for sequential and multicore backends, but we will not be using those here.

A Futhark program consists of a number of top-level definitions, such as functions and constants. Because Futhark is completely pure (apart from non-termination and divergence, e.g. division-by-zero) a function is completely defined in terms of its input parameters. It uses an ML-style syntax reminiscent of Standard ML or Haskell, but with a restricted feature-set that makes it feasible to generate efficient parallel code. Here is an example of a definition of a simple function, *inc*, that increments its input:

```
let inc (x : i64) : i64 =
  x + 1
```

Futhark does not currently support recursive functions (or recursive data-structures). Instead, a semantically sequential loop-construct can be used to imitate tail-recursion. The syntax is:

```
loop x = y for i < z do body
```

The loop executes the body z times. In each iteration of the body, i is bound to the iteration counter between 0 and $z-1$. At the beginning of the first iteration of the loop, x is bound to the value of y; in each following iteration of the loop, x is bound to the result of the previous iteration. Finally, the result of the last iteration is returned. As an example, here is how the factorial function can be defined using `loop`:

```
let fact (n : i64) : i64 =
  loop acc = 1 for i < n do
    acc * (i + 1)
```

2.2.1 Parallelism through Second-Order Array Combinators

Parallelism in Futhark is explicitly expressed by the programmer using a collection of *second-order array combinators*, or SOACs, such as `map`, `reduce` and `scan`, which have the types shown in listing 2.2.

The semantics are conventional: `map` applies a function to each element of an array, `reduce` applies an associative function to successive elements of given an array (starting with a neutral element) and `scan` generalizes `reduce` by collecting each intermediate result.

As an example of how the types should be read, `map` is a function that has two type parameters, $`a$ and $`x$ which are the types of the input and output elements respectively, as well as a size parameter $[n]$, requiring that both the input and output arrays have the same number of elements in them. $[n]a$ is the type of an array of n elements, where each element has type a. Array-types can be nested, so a two-dimensional array would have a type like $[n][m]a$. Notice that the size types prohibit irregular arrays: They cannot be assigned a type in Futhark and are therefore not allowed. The result of the `map` function has the type $^*[n]x$, where the asterisk is used to indicate *uniqueness*. In this case it means that the result of a `map` does not alias any of the inputs, it is entirely *fresh*.

Here is an example of a function, *inc_all*, which increments all the values of the given array using `map`:

```
let inc_all [n] : (xs : [n]i64) : *[n]i64 =
  map inc xs
```

Futhark also supports slicing using conventional triplet notation, which *will* alias its argument, as in the following example:

```
let slice [n] 't (xs : [n]t) (i : i64) : [i]t =
  xs[0 : i]
```

In aggregate, Futhark guarantees that all parallelism expressed by the user is *correct by construction*: Through simple type-checking mechanisms, Futhark guarantees the absence of race conditions, deadlocks and data-races.

Additionally, Futhark's focus on explicit parallelism through SOACs means that the compiler doesn't have to do any complicated analyses in order to extract parallelism from otherwise sequential code. On the other hand, Futhark sometimes has to make decisions about how to *sequentialize* parallel code, since GPUs do not support arbitrary levels of nested parallelism. *Flattening*, the technique used to turn code with arbitrary levels of nested parallelism into something that can run on a GPU is discussed in depth in sections 7.2.1 and 7.2.2.

2.2.2 In-Place Updates

Futhark also supports in-place updates, as in the following two semantically equivalent statements, where the latter is simply syntactic sugar for the former:

```
let ys = ys with [i] = x
let ys[i] = x
```

Note that, although this looks like imperative code, semantically we are not actually updating the value of ys: We are creating a new array whose name

```
let scatter 't [k] [n] : (dest : *[k]t) (is : [n]i64) (vs : [n]t) : *[k]t =
  loop dest = dest for i < n do
    let dest[is[i]] = vs[i]
```

Listing 2.3: The type and sequential semantics of `scatter`.

shadows the old *ys*. However, Futhark's uniqueness types guarantee that the in-place update is safe (the old *ys*, or any aliases thereof, can not be used in any later code path of the program) and that the cost of the in-place update is only proportional to the number of updated values and not to the size of *ys*. In other words, while we are semantically creating a new array, Futhark the implementation is that of a guaranteed safe in-place update.

Futhark also supports another SOAC called `scatter`, which enables parallel in-place updates of an array according to a list of indices. The type and sequential semantics of `scatter` is shown in listing 2.3.

Like for `map`, the result of a `scatter` is unique, but the type of `scatter` also uses an asterisk for one of its arguments, *dest*. This indicates that *dest* is unique as well, meaning that it does not alias any other arrays and is therefore safe to modify in-place. We say that `scatter` now *owns dest*. In this way, uniqueness types in Futhark are vaguely reminiscent of the ownership system in Rust. In short, uniqueness types allow Futhark to support efficient in-place updates of arrays.

`scatter` takes two additional arrays as arguments, *is* and *vs*, and the result is an array containing the values of *dest*, but with all indices contained in *is* replaced by the corresponding values from *vs*. Semantically, we can think of the result as an updated copy of *dest*, but because of the uniqueness types, Futhark can safely reuse the existing array and update the necessary values in-place. Using a work/depth cost-model [BG95], `scatter` has constant *depth* and *work* proportional to the number of updated values, not to the size of the updated array

Chapter 3

Linear Memory Access Descriptors

This chapter describes linear memory access descriptors, or LMADs, and how LMADs can be interpreted both as (1) sets of points in a one-dimensional space used in dependency analysis of arrays, (2) as index functions used to map array elements to memory locations in a compiler IR and (3) as a slicing mechanism for arrays. In addition to laying out the fundamental properties of LMADs and various common operations on them, we also describe a non-overlap test for multi-dimensional LMADs (seen as a set of points), all of which are used extensively in chapters 4 to 6.

3.1 Introduction

Accurately and efficiently aggregating array accesses across loops has long been a topic of research in the context of automatic loop parallelization [RHR03; RPR07; OR13]. Although the traditional triplet-notation of $[\tau:\sigma:\delta,\ldots]$—with τ, σ and δ signifying for each dimension the offset, number of elements and stride—has been used for this purpose, it has been found inadequate for many applications because it cannot accurately describe the access patterns resulting from more complex indexing-operations [PHP98].

As an alternative, and throughout the rest of this thesis, we will use Linear Memory Access Descriptors [Hoe98]. LMADs are superficially similar to the triplet notation, but they make it possible to represent patterns that the triplet-notation does not. An LMAD L, consisting of n dimensions and an offset τ, is written as shown in fig. 3.1. For a given dimension i, σ_i denotes the number of elements of that dimension and must be strictly positive, while δ_i denotes the absolute stride, i.e. the total number of elements separating two consecutive elements of that dimension. Both τ, all σs and all δs must be integers.

A rich body of literature concerning LMADs exists in the context of automatic parallelization of sequential loops in languages like Fortran [RHR03; HPY01;

$$L = \tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_n : \delta_n)\}$$

Figure 3.1: The definition of an LMAD.

$$\{\ \tau + i_1 * \delta_1 + \cdots + i_n * \delta_n \mid 0 \leq i_1 < \sigma_1, \ldots, 0 \leq i_n < \sigma_n\ \}$$

Figure 3.2: The set interpretation of an LMAD $L = \tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_n : \delta_n)\}$.

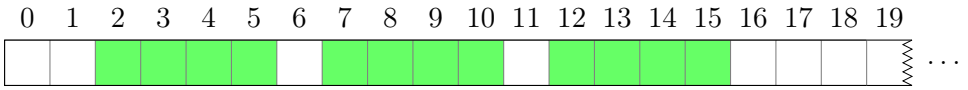


Figure 3.3: The set of points represented by the LMAD $L = 2 + \{(3 : 5), (4 : 1)\}$.

OR15; Rus+06]. In this interpretation, an LMAD corresponds to a set of points in a one-dimensional space given by the equation shown in fig. 3.2. For instance, the LMAD $L = 2 + \{(3 : 5), (4 : 1)\}$ corresponds to the points $\{2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15\}$ as shown in fig. 3.3.

LMADs are more expressive than the traditional triplet notation. For example, they can be used to represent a staggered set of columns as shown in fig. 3.4. This set is given by the LMAD $0 + \{(6 : 9), (3 : 8)\}$.

Apart from the set-interpretation of LMADs, they can also be used as a *slicing mechanism*. Again, the typical notation for slicing an array uses the triplet-notation, but the triplet notation limits what kind of slices we can express. For instance, the triplet notation is restricted to expressing the number of dimensions of the original array (or less), with no ability to create new dimensions. As an example of how LMADs allow you to create new "views" of the existing dimensions of an array, if xs is a one-dimensional array of size mn, the LMAD-slice $xs[\tau + \{(n : m), (m : 1)\}]$ constructs a two-dimensional "view" of the one-dimensional array, akin to unflattening.

Finally, LMADs can be used as *index functions* in the intermediate represen-

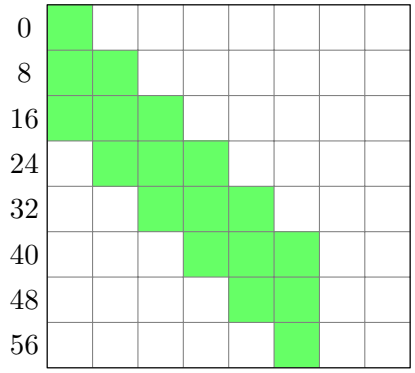


Figure 3.4: A staggered collection of columns.

tation inside a compiler, mapping array elements in the given programming language to specific locations in memory. This allows a compiler to make certain array operations, such as slicing and transposition, "cost-free" for the user: By reusing the memory of the original arrays such operations can be implemented to have zero or negligible runtime cost.

In short, the contributions of this chapter are:

1. A description of how LMADs can be used as a slicing mechanism.
2. A description of LMADs as index functions.
3. A non-overlapping test for multi-dimensional LMADs in the set-of-points representation.
4. A proof-sketch for the correctness of the non-overlapping test.

With the exception of the proof-sketch, the work presented in this chapter has previously either been published at SC22 [Mun+22] or is in print at IFL 2022 [MOHss].

3.2 Background and Related Work

Linear memory access descriptors were first described in 1997 [Pae97] although not named as such until 1998 [PHP98; Hoe98]. In the context of automatic loop parallelization, they are used to aggregate, summarize and reason about cross-iteration dependencies in sequential loops, both in dynamic and static settings [RHR03; RPR07; OR13]. We denote this use of LMADs the *set-interpretation*.

We will also be using the set-interpretation of LMADs when discussing array short-circuiting in chapter 6, but in contrast to the complex operations[1], extensions and runtime support commonly needed for automatic loop parallelization [RHR03; Rus+06; OR13], our use of LMADs is entirely static and we only rely on unions and a non-overlapping test, which we will describe in section 3.6. The non-overlapping test is inspired by the multi-dimensional recursive intersection algorithm described in [HPY01], but builds upon it by applying heuristics to e.g. split dimensions when a given LMAD self-overlaps, leading to less conservative results.

While LMADs under the set interpretation represent a set of points in a one-dimensional space, we can also interpret an LMAD as an index function for an array. An index function is a function $\mathbb{N}^n \to \mathbb{N}$, mapping the values of some n-dimensional space to values in a single dimension. Typically, it is used to map arrays in a given language to concrete offsets in some memory buffer, corresponding to the location in memory of each element of the array.

[1] Such as subtraction and intersection.

A compiler for an array programming language might associate such an index function with each array in the intermediate representation. When the array is later indexed, the index function is applied to find the precise location in memory.

The desire to use LMADs as index functions in the IR of a programming language is closely related to the use of LMADs as a slicing construct. Together, they allow the programmer to use so-called *change-of-layout transformations*, such as slices and transpositions, without the overhead of manifesting the changed arrays to memory, while still allowing the compiler to perform further high-level optimizations on the array uses. As an example, consider the following piece of code:

$$\begin{aligned}
&\texttt{let } xs : [n][m]@xs_{mem} \to f = \ldots \\
&\texttt{let } xs' : [m][n]@xs_{mem} \to transpose(f) = \texttt{transpose } xs \\
&\texttt{let } x : \texttt{int} = xs'[k]
\end{aligned}$$

The first line says that xs is an array with dimensions $n \times m$ residing in the memory buffer xs_{mem} accessed with the index function f. If *transpose* is used to transpose an index function at compile time then xs' can reuse the memory of xs at no runtime cost. Finally, when computing x, we use $(transpose(f))(k)$ to find the corresponding offset in xs_{mem}.

LMADs are closely related to dope vectors, which have been used in implementations for ALGOL 60 [Sat61] and APL [GW78]. The structure of dope vectors are similar to LMADs, but in contrast to LMADs, dope vectors are often used as actual metadata carried around at runtime; in our use, LMADs are strictly a compile-time construct.

3.3 LMADs as a Slice

Many programming languages have some sort of slicing-mechanism for arrays. For instance, Python uses the notation $xs[start:stop:step]$ to indicate the slice of xs starting from element *start* up to *stop* with a step of *step*. For instance the slice $xs[1:10:2]$ would correspond to the values at indices $1, 3, 5, 7, 9$ of xs. An alternative, but isomorphic, notation is $xs[start:number\text{-}of\text{-}elems:step]$, where $xs[1:5:2]$ corresponds to the same indices as above.

However, as we've already seen in fig. 3.4, the triplet notation is not always expressive enough. As a further example, triplet-notation cannot be used to express the blocked anti-diagonal pattern shown in fig. 3.5, which is a simplified version of the pattern used in the NW example discussed extensively in section 6.1. In short, we wish to update the blocks of an anti-diagonal of a matrix in-place, but the slice representing the blocked anti-diagonal cannot be represented using triplet notation. However, by using LMADs as a slicing mechanism and denoting by b the block size, by i the anti-diagonal index and

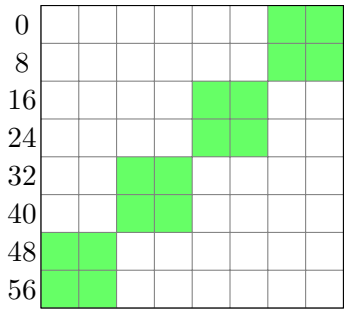


Figure 3.5: The simplified anti-diagonal write pattern from NW.

by n the number of elements per row of the array xs, we can represent the necessary slice as

$$xs[ib + \{(i+1 : nb - b), (b : n), (b : 1)\}]$$

At the most basic level, LMAD-slices make it possible to take a one-dimensional array and unflatten it into an array of higher dimensionality, but it can also be used to represent blocked and skewed views as we have just demonstrated. We will be using LMAD-slicing mostly for one-dimensional arrays, but it can easily be extended to apply to every dimension of an array, as shown in fig. 3.6.

3.4 LMADs as Index Functions

When a programming language supports array slices, the compiler has to decide how to represent those slices at runtime, e.g. where to put the elements of the slice and how the specific address of each element is found. The simplest solution is to always *manifest* all arrays in some known order, usually row-major, a dense format where consecutive elements of a row are located next to each other. When mapping over the slice corresponding to the pattern in fig. 3.5, the values in that slice would first be copied to a new memory buffer in row-major order before applying the map.

Of course, this can be wasteful. After all, the values that are copied to the new memory block already exists in another location, and if we are only reading from the new slice, we might as well reuse the old allocation. In order to do so, we would need to associate with each array an index function, mapping the elements of the array to particular offsets in the associated linear memory space. By using LMADs as index functions, we can model complex slices such as the one from NW with no copying-overhead.

For a q-dimensional LMAD, $L = \tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\}$, we therefore define application as follows:

$$L(y_1, \ldots, y_q) = \tau + y_1\delta_1 + \cdots + y_q\delta_q \qquad \text{(Application)}$$

In other words, to find the memory location of an index $xs[x, y, z]$, we compute $L(x, y, z)$, where L is the index function of xs.

To support our use of LMADs as index functions, we define a set of common operations on LMADs, shown in fig. 3.6.

The *slice*-function is used to slice an LMAD according to an ordinary slice, $[\mathit{offset}\colon \mathit{elems}, \ldots]$. For each dimension, the slice specifies an offset and a number elements. This kind of slicing corresponds closely to regular triplet-notation slices, albeit without the strides, though adding support for strides is trivial, as shown in section 3.4.1.

The alternative slicing function $\mathit{slice}_{\text{LMAD}}$ takes a one-dimensional LMAD L and slices it according to a q-dimensional LMAD, L'. We also define the more general variant, $\mathit{slice}_{\text{LMADs}}$, which is used to slice a multi-dimensional LMAD using multiple LMAD-slices, with each LMAD-slice applied to a separate dimension of the original LMAD.

fix is used to fix the outermost dimension of an LMAD. For instance, given an LMAD $L = 10 + \{(4:4), (4:1)\}$, fixing the outer dimension using $\mathit{fix}(L, 1)$ yields the LMAD $14 + \{(4:1)\}$.

transpose is used to transpose the two outermost dimensions of an LMAD such that $\mathit{transpose}(0 + \{(20:10), (10:1)\}) = 0 + \{(10:1), (20:10)\}$. The result is an element previously at index $[i, j]$ is now at index $[j, i]$, as expected.

Finally, we define $\mathcal{R}(x_1, \ldots, x_n)$ as the row-major LMAD with n dimensions of the sizes indicated, and $\mathcal{C}(x_1, \ldots, x_n)$ as the column-major ditto.

We use these functions to implement zero-cost change-of-layout transformations, as described in detail in chapter 4. In short, any time the user computes a slice or transposition of an array, we are able to reuse the memory of the original array.

3.4.1 Index Function Extensions

For simplicity, especially in later chapters, we have kept the definition and algebra of LMADs described so far simple. For completeness, we will now discuss various extensions to the LMAD algebra that are or have been used, for instance in the Futhark programming language.

Triplet Slices with Strides

First of all, LMADs fully support traditional triplet slices with strides. To handle ordinary triplet slices, we use an extended version of the *slice* function, *slice'*, which takes an LMAD and a slice with $\mathit{offset}\colon \mathit{elems}\colon \mathit{stride}$ triplets:

$$\begin{aligned}&\mathit{slice'}(\tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\}, [x_1 : y_1 : z_1, \ldots, x_q : y_q : z_q]) = \\ &\quad (\tau + \Sigma_i^q x_i \delta_i) + \{(y_1 : z_1 \delta_1), \ldots, (y_q : z_q \delta_q)\}\end{aligned}$$

$$slice(\tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\}, [x_1 : y_1, \ldots, x_q : y_q]) =$$
$$(\tau + \sum_i^q x_i \delta_i) + \{(y_1 : \delta_1), \ldots, (y_q : \delta_q)\} \qquad \text{(Slicing)}$$

$$slice_{\text{LMAD}}(\tau + \{(\sigma : \delta)\}, \tau' + \{(\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\}) =$$
$$\tau + \tau'\delta + \{(\sigma_1 : \delta\delta_1), \ldots, (\sigma_q : \delta\delta_q)\} \qquad \text{(Slicing with an LMAD)}$$

$$\begin{aligned} slice_{\text{LMADS}}(&\tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_n : \delta_n)\}, \\ &\tau_1 + \{(\sigma_{1,1} : \delta_{1,1}), \ldots, (\sigma_{1,m_1} : \delta_{1,m_1})\}, \\ &\qquad\qquad \vdots \\ &\tau_n + \{(\sigma_{n,1} : \delta_{n,1}), \ldots, (\sigma_{n,m_n} : \delta_{n,m_n})\}) = \\ &\tau + \tau_1\delta_1 + \cdots + \tau_n\delta_n + \\ &\{(\sigma_{1,1} : \delta_1\delta_{1,1}), \ldots, (\sigma_{1,m_1} : \delta_1\delta_{1,m_1}), \\ &\qquad\qquad \vdots \\ &(\sigma_{n,1} : \delta_n\delta_{n,1}), \ldots, (\sigma_{n,m_n} : \delta_n\delta_{n,m_n})\} \end{aligned} \qquad \text{(Slicing with multiple LMADs)}$$

$$fix(\tau + \{(\sigma_1 : \delta_1) \cdots (\sigma_q : \delta_q)\}, k) =$$
$$(\tau + k \cdot \delta_1) + \{(\sigma_2 : \delta_2) \cdots (\sigma_q : \delta_q)\} \qquad \text{(Fix)}$$

$$transpose(\tau + \{(\sigma_1 : \delta_1), (\sigma_2 : \delta_2), \ldots, (\sigma_q : \delta_q)\}) =$$
$$\tau + \{(\sigma_2 : \delta_2), (\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\} \qquad \text{(Transpose)}$$

$$\mathcal{R}(x_1, \ldots, x_n) = 0 + \{(x_1 : \prod_{i=2}^{n} x_i), \ldots, (x_n : 1)\} \qquad \text{(Row-Major)}$$

$$\mathcal{C}(x_1, \ldots, x_n) = 0 + \{(x_1 : 1), \ldots, (x_n : \prod_{i=1}^{n-1} x_i)\} \qquad \text{(Column-Major)}$$

Figure 3.6: Common operations on LMADs.

Dimension Permutation

Similarly, while we have described a *transpose* function which transposes the two outer-most dimensions of an LMAD, it is possible to extend the algebra with a more general *permute* function. In Futhark, permutation of array dimensions is not directly represented in the source language, but when the source code is translated to an intermediate representation, (nested) calls to *transpose* is translated to a *permute* primitive. Given an LMAD and a list of indices $[i_0, \ldots, i_k]$, the definition of *permute* is:

$$\begin{aligned} &\mathit{permute}(\tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\}, [i_0, \ldots, i_k]) = \\ &\quad \tau + \{(\sigma_{i_0} : \delta_{i_0}), \ldots, (\sigma_{i_k} : \delta_{i_k})\} \end{aligned}$$

In other words, *permute* permutes the dimensions of a given LMAD according to the given indices. However, because the Futhark compiler will sometimes be able to extract useful information from the permutation of an array in order to perform later optimizations, each dimension in the Futhark version of LMADs actually has an additional permutation parameter, π, indicating the permuted index of the given dimension. The resulting representation looks like this:

$$\tau + \{(\sigma : \delta : \pi), \ldots\}$$

In order to compute the "effective" LMAD dimensions, we must first permute them according to the π-values. For an LMAD with permutations, application is performed by first permuting the dimensions such that they are ordered by their permutation values and then performing regular application.

Rotation

A common operation on arrays is rotation: Given an array $x = [0, 1, 2]$, `rotate`$(x, 1)$ produces the array $[1, 2, 0]$. It is possible to support rotates in the IR without manifestation by annotating LMADs with an additional rotation factor for each dimension, ρ. The resulting representation looks like this:

$$\tau + \{(\sigma : \delta : \rho), \ldots\}$$

Our LMAD application function needs to be amended to take rotations into account:

$$L(y_1, \ldots, y_q) = \tau + \Sigma_{i=1}^{i<q}((y_i + \rho_i) \bmod \sigma_i) * \delta_i \qquad (\text{Application}_{\text{rot}})$$

Futhark did have support for rotations in LMADs for a while, but it has been removed in order to simplify the IR.

Compositions of LMADs

Besides the added complexity and the fact that modulo and division could be costly at runtime, adding rotations to the algebra also means that the algebra is no longer closed under composition: Taking a slice of a rotated LMAD cannot always be represented as an LMAD. For instance, given the LMAD $L = 0 + \{(10:1:5)\}$, taking the $[3:4]$ slice should result in an LMAD representing the flat offsets $[8, 9, 0, 1]$, but that is not possible using a one-dimensional LMAD.

To alleviate this situation, we can change the index function representation to consist of a list of LMADs instead of a single LMAD. The sliced and rotated index function from above would then be represented as the list $[3 + \{(4:1:0)\}, 0 + \{(10:1:5)\}]$. Application of index functions with multiple LMADs is done by applying the first LMAD to get a flat offset, then unranking that index by the shape of the second LMAD and finally applying the unranked index to the second LMAD, as seen in proc. 3.1[2]. Unranking is done by taking a flat offset and finding the multi-dimensional index of an LMAD that would correspond to that flat offset in a row-major representation, as shown in proc. 3.2. For example, given the flat offset 10 and the two-dimensional LMAD $L = 0 + \{(4:4), (4:1)\}$, the unranked index is $[2, 2]$. Unranking an index at run-time involves expensive division and modulo operations, but does not occur often in actual code.

Procedure 3.1: ApplyLMADs$(L_1, L_2, \bar{i})$

input : LMADs $L_1 = \tau_1 + \{(\sigma_1^1 : \delta_1^1), \ldots, (\sigma_n^1 : \delta_1^1)\}$, $L_2 = \tau_2 + \{(\sigma_1^2 : \delta_1^2), \ldots, (\sigma_m^2 : \delta_m^2)\}$ and an n-dimensional index $[i_1, \ldots, i_n]$.

output : The corresponding index into L_2.

```
1 x ⟵ L_1(i_1, ..., i_n);
2 j_1, ..., j_m ⟵ Unrank(L_2, x);
3 y ⟵ L_2(j_1, ..., j_m);
4 return y;
```

Reshape

Reshaping is the process of taking an LMAD, flattening it to a one-dimensional view and applying a new LMAD on top. For instance:

$$\mathit{reshape}(10 + \{(20:10), (10:1)\}, 0 + \{(50:4), (4:1)\}) = 10 + \{(50:4), (4:1)\}$$

Like rotation, reshaping an array can lead to index functions that are not representable by a single LMAD, and therefore often leads to an LMAD composition.

[2] ApplyLMADs only applies to two LMADs, but can be generalized to an arbitrary amount of LMADs.

Procedure 3.2: UNRANK(L, x)

input : An LMAD $\tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_n : \delta_n)\}$, and a flat offset x.
output : An n-dimensional index corresponding to the unranked x.

$\overline{slc} \longleftarrow \bullet$;
for $i < n$ **do**
  $tmp \longleftarrow \prod_{j=i+1}^{j \leq n} \sigma_j$;
  Append $\frac{x}{tmp}$ to $\overline{slc}$;
  $x \longleftarrow x \bmod tmp$;
return $\overline{slc}$;

3.5 LMADs as Sets of Points

We can also interpret an LMAD as representing a set of points in a one-dimensional space, using the definition from fig. 3.2, repeated here for reference:

$$\{\ \tau + i_1 * \delta_1 + \cdots + i_n * \delta_n \mid 0 \leq i_1 < \sigma_1, \ldots, 0 \leq i_n < \sigma_n\ \}$$

This is the common interpretation, and as already described it is useful when we want to reason about array accesses, for instance when performing automatic parallelization or array short-circuiting, as in chapter 6.

Under this interpretation, an LMAD, L, has a *domain*, $dom(\mathrm{L})$, constituting all valid indices as well as an *image*, $img(\mathrm{L})$, constituting all addresses that are reachable by applying indices within the domain. Zero-dimensional LMADs correspond to a single point in space.

3.5.1 Image-Preserving Operations

Under the set-interpretation of LMADs, we can freely modify a given LMAD as long as we do not change its image. This can be helpful when using LMADs for further analysis and optimizations. We now present a series of operations that are only safe when regarding LMADs as sets of points.

Normalization

If we statically know the ordering and sign of the strides of an LMAD, we can always *normalize* the LMAD. Normalizing an LMAD turns it into another LMAD with only positive strides and dimensions sorted by stride. The normalized LMAD has the same image as the original LMAD, but the corresponding index function or slice has changed. Normalization therefore only applies when we are using the set-interpretation of LMADs.

Pseudocode for the normalization procedure can be seen in proc. 3.3. For an LMAD, $L = \tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_n : \delta_n)\}$, normalizing the negatively strided dimension at index i is done by negating the stride of that dimension and

adding $\sigma_i\delta_i + 1$ to the global offset. Afterwards, the dimensions are sorted by their normalized strides. As an example the LMAD $L = 16 + \{(4:-1),(8:4)\}$ is normalized to $L' = 13 + \{(8:4),(4:1)\}$. When reasoning about the sets represented by an LMAD, we will for the most part assume that it has already been normalized.

Procedure 3.3: NORMALIZE(L)

input : An LMAD $L = \tau + \{(\sigma_1:\delta_1),\ldots,(\sigma_n:\delta_n)\}$.
output : The normalized LMAD.

```
1 res ⟵ •;
2 for i < n do
3     if δ_i < 0 then
4         τ ⟵ τ + σ_i δ_i + 1;
5         Append (σ_i : −δ_i) to res;
6     else
7         Append (σ_i : δ_i) to res;
8 res ⟵ sort res in decreasing order by stride;
9 return τ + {res};
```

Inserting Empty Dimensions

It is always possible to insert "empty" dimensions anywhere in an LMAD without changing the image of the LMAD. An empty dimension is a dimension with any stride and $\sigma = 1$: The only valid index for that dimension is 0, so no matter what stride is used it holds that $0 \cdot \delta = 0$. Therefore the extra dimension does not change the image of the LMAD. For instance, given the LMAD $L = \tau + \{(\sigma_1:\delta_1)\}$, we can insert an empty dimension at the end to get $L' = \tau + \{(\sigma_1:\delta_1),(1:1)\}$

Splitting and Joining Dimensions

Given an LMAD L containing two dimensions $(\sigma_i:\delta_i)$ and $(\sigma_j:\delta_j)$ such that $\delta_i = \sigma_j\delta_j$, we can join the two dimensions without changing the image of the LMAD by removing the two old dimensions and inserting a new with number of elements $\sigma_i\sigma_j$ and stride δ_j:

$$\begin{aligned} &\mathit{join}(\tau + \{\ldots,(\sigma_i:\sigma_j\delta_j),(\sigma_j:\delta_j),\ldots\}, i) = \\ &\quad \tau + \{\ldots,(\sigma_i\sigma_j:\delta_j),\ldots\} \end{aligned} \tag{Join}$$

For instance: $\mathit{join}(\tau + \{(20:10),(10:1)\}, 1) = \tau + \{(200:1)\}$.

Similarly, given an LMAD L with a dimension $(\sigma_i:\delta_i)$ and a split j that divides σ_i, we can always split that dimension into two new dimensions as

follows:

$$\begin{aligned} &\mathit{split}(\tau + \{\ldots, (\sigma_i : \delta_i), \ldots\}, i, j) = \\ &\quad \tau + \{\ldots, (\frac{\sigma_i}{j} : j\delta_i), (j : \delta_i), \ldots\} \end{aligned} \tag{Split}$$

The result is that we can reverse a join, e.g.: $\mathit{split}(\tau + \{(200:1)\}, 1, 10) = \tau + \{(20:10), (10:1)\}$.

3.5.2 LMAD Expansion

Just as we can reduce the number of dimensions of an LMAD using *fix*, it can be helpful to be able to *expand* an LMAD to increase the number of dimension as well. For instance, when analyzing a sequential loop containing an array access, we would like to be able to summarize the accesses into one LMAD. Consider the following example, where xs is an array with index function L:

`for` $i \leq n$ `do`	What is the aggregated access?
$\ldots xs[i] \ldots$	Access with LMAD $L(i)$

Procedure 3.4: EXPAND(i, l, s, L)

input : A lower bound l, a number of elements (or span) s and an iterator variable i, such that $l \leq i < l + s$, as well as an LMAD $L = \tau + \{\overline{d}\}$.

output : The LMAD expanded with a new dimension.

if *i is free in $\overline{d}$* **then**
  fail
$k \longleftarrow fresh$;
$\mathit{offset} \longleftarrow \tau[k/i]$;
$\mathit{offset'} \longleftarrow \tau[k+1/i]$;
$\mathit{newStride} \longleftarrow \mathit{offset'} - \mathit{offset}$;
$\mathit{newOffset} \longleftarrow \tau[l/i]$;
$L' \longleftarrow \mathit{newOffset} + \{(s : \mathit{newStride}), \overline{d}\}$;
if *k is free in L'* **then**
  fail
else
  return L';

To aggregate the accesses across the loop, we would like a systematic way to reverse *fix*. Procedure 3.4 shows the pseudo-code for EXPAND, which takes an iterator variable, i, a lower bound, l, a number of elements, s and an LMAD, L, and "expands" the LMAD by adding a new dimension. It works by first asserting that i does not occur in any of the dimensions in L. Then, it creates a

fresh name k, and computes *offset* and *offset'* by replacing the iterator variable inside the LMAD-offset τ with k and $k+1$, respectively. The new stride is computed by taking the difference between *offset'* and *offset*, while the new offset is computed by replacing i with the lower bound l in τ. Finally, we assert that our temporary variable k does not occur in the resulting LMAD L' once it has been simplified.

Using EXPAND, we can aggregate array accesses across recurrences such as loops and maps. In fact, expanding an LMAD over a loop corresponds to taking the union of the LMADs representing the array accesses of each iteration. Therefore, if the ith iteration of a loop has array accesses that can be summarized by L_i, l is the lower bound of the iterator variable (usually 0) and s is the upper bound, then:

$$\bigcup_{i=l}^{s+l} L_i \approx \text{EXPAND}(i, l, s, L)$$

Sometimes expansion will not work, e.g. if the iterator variable is used in any of the dimensions of the LMAD to expand. In this case, we can always use a *set of* LMADs, to represent the union of their accesses. In other words:

$$\bigcup_{i=l}^{s+l} L_i = \{L_i \mid l \leq i \leq s+l\}$$

3.6 LMAD Overlap

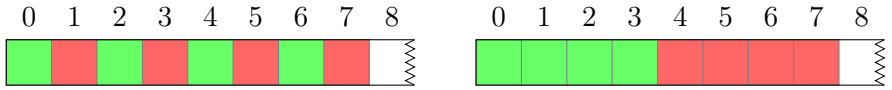
When reasoning about recurrent array accesses using LMADs, we will often need to decide whether two given LMADs *overlap*. We'll start by describing the simple case in which each LMAD has a single dimension, before moving on to the more general case.

3.6.1 One-dimensional LMADs

For two one-dimensional LMADs, L_1 and L_2, there are three cases for which they do not overlap: Either all points in L_1 come before all points in L_2, all points in L_2 come before all points in L_1 or the points of the two LMADs are interleaved in memory. Figure 3.7 shows two examples: In fig. 3.7a the accesses of the two LMADs (colored green and red, respectively) are interleaved, while in fig. 3.7b the accesses of one LMAD are all located before the accesses of the other.

As a result, non-overlap for two one-dimensional LMADs with positive strides, $\tau_1 + \{(\sigma_1 : \delta_1)\}$ and $\tau_2 + \{(\sigma_2 : \delta_2)\}$, can be checked by the predicate [OR12]:

$$\gcd(\delta_1, \delta_2) \nmid (\tau_1 - \tau_2) \ \vee \ \tau_1 > \tau_2 + \sigma_2 \ \vee \ \tau_2 > \tau_1 + \sigma_1$$



(a) Interleaved accesses. **(b)** Disjoint accesses.

Figure 3.7: Possible cases for non-overlapping LMADs in one dimension.

3.6.2 Multi-dimensional LMADs

Multi-dimensional LMADs pose some additional problems when attempting to statically prove non-overlap. Most notably, LMADs can have overlapping dimensions and we might want to be able to compare LMADs with different strides and numbers of dimensions. A naive solution would be to use formula given in fig. 3.2 for the two LMADs of interest and then attempt to prove that the resulting sets are disjoint. However, that is infeasible, probably intractable, for even slightly complex examples[3] [HPY01].

Instead, we present an alternative heuristic for statically determining whether two LMADs overlap or not, based on an *interval representation*, called a *sum of strided intervals*, or SOSI. The heuristic is conservative: If any step along the way cannot be computed statically, we fail and report that the two LMADs may overlap. In other words, we do not claim to be able to handle all LMADs, but we have found this heuristic good enough to prove non-overlap for the NW and LUD benchmarks, as described in chapter 6.

The process to determine whether two LMADs L_1 and L_2 have an empty intersection, i.e. do not overlap, is as follows:

1. Normalize each LMAD.
2. Turn each LMAD into a SOSI.
3. Match up the strides of the SOSIs by inserting empty dimensions as necessary while preserving normalization, e.g. the dimensions of the resulting SOSIs should still be ordered by stride.
4. Distribute the offsets of the SOSIs, such that each resulting SOSI has an offset of zero.
5. Determine whether either of the SOSIs have self-overlapping dimensions. If they do, attempt to split the offending dimension and return to step 3.
6. If both SOSIs have no self-overlapping dimensions, determine whether the two SOSIs contain a pair of dimensions with matching strides that does not overlap. If successful, the LMADs do not overlap.

Procedure 3.5 shows the pseudo-code corresponding to steps 3–6.

[3] And indeed, we attempted to use an external SMT solver [MB08] to prove non-overlap of our NW problem, but it was not able to do so.

Procedure 3.5: NoOverlap(I_1, I_2)

input : Two normalized SOSIs, I_1 and I_2.
output : **true** if $I_1 \cap I_2 = \emptyset$ can be proved, **false** otherwise

1 $I_1', I_2' \longleftarrow$ match up strides of I_1 and I_2 by inserting empty dimension and preserving normalization;
2 $[l_1^1, u_1^1]\delta_1 + \cdots + [l_n^1, u_n^1]\delta_n,\ [l_1^2, u_1^2]\delta_1 + \cdots + [l_n^2, u_n^2]\delta_n \longleftarrow$ distribute offsets of I_1' and I_2';
3 **if** *there exists i such that* $\delta_i \leq u_{i+1}^1\delta_{i+1} + \cdots + u_n^1\delta_n$ **then**
4 $\quad I_1^1 \longleftarrow [l_1^1, u_1^1]\delta_1 + \cdots + [l_i^1, u_i^1 - 1]\delta_i + \cdots + [l_n^1, u_n^1]\delta_n$;
5 $\quad I_1^1 \longleftarrow u_i^1\delta_i + [l_1^1, u_1^1]\delta_1 + \cdots + [l_{i-1}^1, u_{i-1}^1]\delta_{i-1} + [l_{i+1}^1, u_{i+1}^1]\delta_{i+1} + \cdots + [l_n^1, u_n^1]\delta_n$;
6 $\quad$ **return** NoOverlap$(I_1^1, I_2'') \wedge$ NoOverlap(I_1^2, I_2'');
7 **else if** *there exists i such that* $\delta_i \leq u_{i+1}^2\delta_{i+1} + \cdots + u_n^2\delta_n$ **then**
8 $\quad$ Symmetrical;
9 **else**
10 $\quad$ **if** *there exists i such that* $I_1'' \cap I_2'' = \emptyset$ **then**
11 $\quad\quad$ **return true**;
12 $\quad$ **else**
13 $\quad\quad$ **return false**;

Each step along the way preserves the image of the resulting SOSIs such that they match the image of the original LMADs.

The first step is straightforward: Normalizing LMADs is described in section 3.5.1. Of course, we may not be able to statically determine the correct ordering of the strides, in which case the analysis should conservatively fail.

The next step is to convert the LMADs into SOSIs. A SOSI consists of a global offset, τ, and a sum of intervals with strides of the form, $[l, u] \cdot \delta$. Like an LMAD, a SOSI represents a set of points in a one-dimensional space as defined by the following formula, where we require both offsets, strides and interval bounds to be non-negative:

$$\tau + \Sigma_{i=1}^m [l_i, u_i] \cdot \delta_i = \{\tau + \Sigma_{i=1}^m j_i\delta_i \mid l_1 \leq j_1 \leq u_1, \ldots, l_m \leq j_m \leq u_m\} \quad (3.1)$$

In order to turn two LMADs into SOSIs, we use *toInterval* as defined in fig. 3.8. Then we can insert extra empty dimensions such that the two SOSIs have dimensions with matching strides, while being careful to preserve the normalization of the resulting SOSIs.

Distributing Offsets (step 4)

To get rid of the global offsets, we need to distribute the terms of each offset among the intervals of the SOSIs, which we can do without changing the image

$$
\begin{aligned}
&\mathit{toInterval}(\tau + \{(\sigma_1 : \delta_1), \ldots, (\sigma_q : \delta_q)\}) = \\
&\quad \tau + [0, \sigma_1 - 1] \cdot \delta_1 + \ldots + [0, \sigma_q - 1] \cdot \delta_q
\end{aligned}
$$

Figure 3.8: The definition of *toInterval*.

of the SOSIs. Distributing the offset requires a bit of symbolic algebra support for simplifying and distributing terms to different intervals. Let's look at an example. Consider two normalized SOSIs with matching strides:

$$
\begin{aligned}
I_1 &= \tau_1 + [l_1^1, u_1^1] \cdot \delta_1 + \cdots + [l_n^1, u_n^1] \cdot \delta_n \\
I_2 &= \tau_2 + [l_1^2, u_1^2] \cdot \delta_1 + \cdots + [l_n^2, u_n^2] \cdot \delta_n
\end{aligned}
$$

We wish to get rid of the offsets τ_1 and τ_2 by distributing them onto the intervals of the SOSIs. To do so, we attempt to match up the terms of the intervals with the strides of the dimensions that best match it. For instance, if for a given SOSI the offset is equal to one of the strides, e.g. $\delta_1 + [l_1, u_1] \cdot \delta_1 + \cdots$, we can distribute the offset onto the interval with stride δ_1 without changing the image of the SOSI:

$$
\delta_1 + \delta_2 [l_1, u_1] \cdot \delta_1 = [l_1 + 1, u_1 + 1] \cdot \delta_1
$$

When distributing across two SOSIs, $I_1 = \tau_1 + \ldots$ and $I_2 = \tau_2 + \ldots$, we can simplify matters a bit, by subtracting τ_2 from τ_1 and add the positive terms to intervals in I_1 and add the absolute value of the negative terms to intervals in I_2. This changes the image of the SOSIs, but we claim that I_1 and I_2 overlap if and only if the two resulting SOSIs I_1' and I_2' overlap.

Proof. Given two SOSIs, I_1 and I_2, where τ_1^+ and τ_1^- are the positive and negative terms of the offset of I_1, and vice versa for I_2:

$$
\begin{aligned}
I_1 &= \tau_1^+ - \tau_1^- + [l_1^1, u_1^1]\delta_1 \cdots [l_n^1, u_n^1]\delta_n \\
I_2 &= \tau_2^+ - \tau_2^- + [l_1^2, u_1^2]\delta_1 \cdots [l_n^2, u_n^2]\delta_n
\end{aligned}
$$

Assume there exists $i_1, \ldots, i_n$ and $j_1, \ldots, j_n$ in range of the corresponding dimensions of each SOSI such that:

$$
\tau_1^+ - \tau_1^- + i_1\delta_1 \cdots i_n\delta_n = \tau_2^+ - \tau_2^- + j_1\delta_1 \cdots j_n\delta_n
$$

Then it also holds that

$$
\tau_1^+ + \tau_2^- + i_1\delta_1 \cdots i_n\delta_n = \tau_2^+ + \tau_1^- + j_1\delta_1 \cdots j_n\delta_n
$$

Therefore, we can split up and distribute the negative and positive terms of the offsets without affecting whether the two SOSIs overlap. □

Things become more complicated when the terms of the offsets are not exactly equal to the strides of the intervals. Consider the following case, where n and b are known to be positive:

$$I = nb + [l_1, u_1] \cdot (nb - b) + [l_2, u_2] \cdot n + [l_3, u_3] \cdot 1$$

The offset is nb. Of course, any offset is divisible by the stride of the last dimension, 1, but nb is more closely related to the stride of the first dimension, $nb - b$. In order to apply it there, we rewrite $\tau = nb = (nb - b) + b$. Then, we directly distribute $nb - b$ to the first dimension and b to the last, yielding:

$$\begin{aligned} I &= nb + [l_1, u_1] \cdot (nb - b) + [l_2, u_2] \cdot n + [l_3, u_3] \cdot 1 \\ &= (nb - b) + b + [l_1, u_1] \cdot (nb - b) + [l_2, u_2] \cdot n + [l_3, u_3] \cdot 1 \\ &= b + [l_1 + 1, u_1 + 1] \cdot (nb - b) + [l_2, u_2] \cdot n + [l_3, u_3] \cdot 1 \\ &= [l_1 + 1, u_1 + 1] \cdot (nb - b) + [l_2, u_2] \cdot n + [l_3 + b, u_3 + b] \cdot 1 \end{aligned}$$

Once again, this process may fail, e.g. if we are not be able to statically determine whether the terms of the offsets are positive or negative. In this case, the non-overlapping test conservatively fails.

No Self-Overlap (step 5)

Having transformed the SOSIs in order to get rid of the offsets, we now need to make sure that each SOSI does not overlap itself. Self-overlapping happens when the span of lower dimensions is larger than the stride of a larger dimension. For example, the following SOSI self-overlaps, because the total stride of first dimension (10) is not strictly larger than the span of the rest of the SOSI (10):

$$[0, 1] \cdot 10 + [0, 10] \cdot 1$$

For a given normalized SOSI with zero offset $I_1 = [l_1, u_1] \cdot \delta_1 + \cdots + [l_n, u_n] \cdot \delta_n$, the following condition is sufficient for I to have no self-overlapping dimensions:

$$\forall i.\ \delta_i > \sum_{k=i+1}^{n} u_k \delta_k \tag{3.2}$$

If a SOSI is found to be self-overlapping, e.g. there is a particular dimension i such that $\delta_i \leq \sum_{k=i+1}^{n} u_k \delta_k$, we split up the dimension with index $i + 1$, resulting in two new SOSIs. For instance:

$$[0, 1] \cdot 10 + [0, 10] \cdot 1 = \underbrace{[0, 1] \cdot 10 + [0, 9] \cdot 1}_{I_1'} \cup \underbrace{10 + [0, 1] \cdot 10}_{I_1''}$$

Assuming that I_2 is some other SOSI, it must hold that if both $I_1' \cap I_2 = \emptyset$ and $I_1'' \cap I_2 = \emptyset$, then $I_1 \cap I_2 = \emptyset$ is also true. We therefore recursively continue the non-overlapping test with these two new SOSIs. To guarantee termination, we impose an upper bound to how many times we recursively attempt to split the dimensions of a SOSI.

The Non-Overlap Test Between Two SOSIs (step 6)

Having transformed our LMADs into SOSIs, I_1 and I_2, with matching strides, no offset and no self-overlap, it follows that $I_1 \cap I_2 \neq \emptyset$ if and only if there exists some indices for I_1 and I_2, $\bar{j}^1$ and $\bar{j}^2$ respectively, for which it holds that $\forall i.\ l_i^1 \leq j_i^1 \leq u_i^1 \ \wedge \ l_i^2 \leq j_i^2 \leq u_i^2$ and that:

$$\sum_{i=1}^{n} j_i^1 \delta_i = \sum_{i=1}^{n} j_i^2 \delta_i \tag{3.3}$$

But if that is true, then $\bar{j}^1$ must be exactly equal to $\bar{j}^2$, as per the following lemma.

Lemma 3.6.1. *If $I_1 \cap I_2 \neq \emptyset$ and $\bar{j}^1$ and $\bar{j}^2$ are valid indices for I_1 and I_2 that satisfy eq.* (3.3)*, then*

$$\forall i.\ j_i^1 = j_i^2$$

Proof. Assume that there exists minimal q such that $j_q^1 \neq j_q^2$. Without loss of generality, assume that $j_q^1 > j_q^2$. We then have that:

$$\begin{aligned}
(j_q^1 - j_q^2)\delta_q &\geq \delta_q && \text{Because } j_q^1 > j_q^2 \\
&> \sum_{k=q+1}^{n} u_k^2 \delta_k && \text{Because there is no self-overlap} \\
&\geq \sum_{k=q+1}^{n} (j_k^2 - j_k^1)\delta_k && \text{Because } u_k^2 \geq j_k^2 \text{ and } j_k^1 \geq 0
\end{aligned}$$

Implying that

$$
\begin{aligned}
& (j_q^1 - j_q^2)\delta_q > \sum_{k=q+1}^{n} (j_k^2 - j_k^1)\delta_k \\
\Leftrightarrow \quad & \sum_{k=q}^{n} (j_k^1 - j_k^2)\delta_k > 0 \\
\Leftrightarrow \quad & \sum_{k=q}^{n} j_k^1 \delta_k > \sum_{k=q}^{n} j_k^2 \delta_k \\
\Leftrightarrow \quad & \sum_{k=1}^{n} j_k^1 \delta_k > \sum_{k=1}^{n} j_k^2 \delta_k \qquad \text{Because } q \text{ is minimal}
\end{aligned}
$$

This contradicts the assertion in eq. (3.3). □

This lemma lets us prove the following theorem, which states the sufficient condition we will use to show non-overlap of SOSIs.

Theorem 3.6.2 (SOSI Non-Overlap)**.** *Given two normalized* SOSI*s, I_1 and I_2 with zero offsets and no self-overlap, a sufficient condition for $I_1 \cap I_2 = \emptyset$ is that there is at least one pair of intervals which do not overlap, e.g.:*

$$\exists i.\ [l_i^1, u_i^1] \cap [l_i^2, u_i^2] = \emptyset \text{ or equivalently } u_i^2 < l_i^1 \vee u_j^1 < l_2^2 \tag{3.4}$$

Proof. Assume that there is a dimension i in I_1 and I_2 for which the intervals di not overlap, as in eq. (3.4). Then by lemma 3.6.1 it cannot be the case that $I_1 \cap I_2 \neq \emptyset$, because there exists no j_i^1 and j_i^2 within the valid bounds that satisfy $j_i^1 = j_i^2$. □

As an example, given the two SOSIs $I^1 = [0,2] \cdot 10 + [0,4] \cdot 1$ and $I^2 = [0,2] \cdot 10 + [5,9] \cdot 1$, we can see that even though the first pair of dimensions overlap, the second pair does not, so the points represented by those intervals do not overlap. We can validate this assertion by enumerating all the points of each SOSI:

$$
\begin{aligned}
I^1 &= \{0,1,2,3,4,10,11,12,13,14,20,21,22,23,24\} \\
I^2 &= \{5,6,7,8,9,15,16,17,18,19,25,26,27,28,29\}
\end{aligned}
$$

Apart from some symbolic algebra in handling the splitting of dimensions, all of the tests involved in proving non-overlap of LMADs come down to relatively simple inequalities.

3.6.3 Example: Proving Non-Overlap of NW

Using the NW example, which is explained in more detail in sections 6.1 and 6.8, we now show an example of how non-overlap is proven.

Assuming that

$$n = qb + 1,\ q > 1,\ b > 1,\ i \geq 1$$

we wish to prove that the following two LMADs, L_W and L_R, do not overlap, e.g. $L_W \cap L_R = \emptyset$:

$$\begin{aligned} L_W &= ib + n + 1 + \{(i+1 : nb - b), (b : n), (b : 1)\} \\ L_R &= ib + \{(i+1 : nb - b), (b+1 : n)\} \end{aligned}$$

The LMADs have already been normalized, so we start by transforming them into SOSIs and inserting extra dimensions to match the strides:

$$\begin{aligned} I_W &= ib + n + 1 + [0,i] \cdot (nb - b) + [0, b-1] \cdot n + [0, b-1] \cdot 1 \\ I_R &= ib + [0,i] \cdot (nb - b) + [0,b] \cdot n \\ &= ib + [0,i] \cdot (nb - b) + [0,b] \cdot n + [0,0] \cdot 1 \end{aligned}$$

Next, we need to distribute the offsets. By subtracting the offset of I_R from the offset of I_W we get $n + 1$, which are added to the intervals of I_W, yielding:

$$\begin{aligned} I'_W &= [0,i] \cdot (nb - b) + [1,b] \cdot n + [1,b] \cdot 1 \\ I'_R &= [0,i] \cdot (nb - b) + [0,b] \cdot n + [0,0] \cdot 1 \end{aligned}$$

Both I'_W and I'_R have self-overlapping dimensions, since the stride of the first interval is not greater than the span of the later dimensions, because $nb - b \not> nb$. Therefore, we split the second dimension, e.g. $[0,b] \cdot n = [0, b-1] \cdot n \cup \{nb\}$. The result is four new SOSIs, some of which have new offsets we need to distribute:

$$\begin{aligned} I^1_W &= nb + [0,i] \cdot (nb - b) + [0,0] \cdot n + [1,b] \cdot 1 \\ &= (nb - b) + b + [0,i] \cdot (nb - b) + [0,0] \cdot n + [1,b] \cdot 1 \\ &= [1, i+1] \cdot (nb - b) + [0,0] \cdot n + [b+1, 2b] \cdot 1 \end{aligned}$$

$$\begin{aligned} I^2_W &= [0,i] \cdot (nb - b) + [1, b-1] \cdot n + [1,b] \cdot 1 \\ I^1_R &= nb + [0,i] \cdot (nb - b) + [0,0] \cdot n + [0,0] \cdot 1 \\ &= (nb - b) + b + [0,i] \cdot (nb - b) + [0,0] \cdot n + [0,0] \cdot 1 \\ &= [1, i+1] \cdot (nb - b) + [0,0] \cdot n + [b,b] \cdot 1 \end{aligned}$$

$$I^2_R = [0,i] \cdot (nb - b) + [0, b-1] \cdot n + [0,0] \cdot 1$$

None of the resulting SOSIs have self-overlapping dimensions, so we can apply the test from the non-overlap theorem. To prove $(I^1_W \cup I^2_W) \cap (I^1_R \cup I^2_R) = \emptyset$, it is enough to prove that they are pairwise disjoint.

We get that $W_I^1 \cap R_I^1 = \emptyset$, $W_I^1 \cap R_I^2 = \emptyset$ and $W_I^2 \cap R_I^2 = \emptyset$ because the last dimensions do not overlap, e.g. $[b+1, 2b] \cap [b, b] = \emptyset$.

Likewise, $W_I^2 \cap R_I^1 = \emptyset$ because the second dimension does not overlap. In other words $[1, b-1] \cap [0, 0] = \emptyset$.

Chapter 4

Memory in a Functional Language IR

In this chapter, we present a technique for introducing a notion of memory to the IR of a functional parallel array-oriented language. We show how the resulting IR, FunMem, which uses LMADs as index functions, lets us reason about and optimize memory access patterns and usage while staying in a high-level value-based semantics.

4.1 Introduction

In functional array languages, computations based on multi-dimensional arrays are expressed in terms of *second-order array combinators* and *change-of-layout* operations, as described in more detail in section 2.2. SOACs, such as `map`, `reduce`, `scan` and `scatter`, take arrays as inputs and produce new arrays as outputs. Change-of-layout operations create new arrays by reordering, reshaping or selecting a subset of values from already existing arrays. Expressing computations in terms of these combinators and operations lends itself well to parallelization, and indeed, a promising line of research uses parallel functional languages for efficient GPU execution [Hen+19].

One challenge for these languages, is that GPU performance is highly sensitive to choices around how arrays are allocated and mapped to memory. In pure and functional languages, there is no notion of memory that the user can use to express memory layout optimizations themselves, so it is up to the compiler to determine the best way to represent different arrays. Traditionally, memory is introduced by the compiler when translating the functional intermediate representation (IR) to an imperative IR, but that means that the compiler loses the ability to reason about memory at a higher, functional level.

In this chapter, we present a technique for representing arrays and memory mappings in the intermediate representation of a functional parallel array

language, without losing the ability to reason about memory optimizations at a high level. The idea is to take the functional IR and amend it with support for allocations and by associating with each array an index-function, both inserted automatically by the compiler. The resulting IR still has a value-based semantics that we can use to perform high-level optimizations, but it also has a memory-oriented semantics that we can use to optimize memory usage.

As an example, consider the following bit of code in our functional IR, FUN, where a two-dimensional array xs[1] is sliced to compute xs' (`kernel` is a parallel loop, described in more detail in section 4.3):

$$\texttt{let } xs : [n][m] = \texttt{kernel } \ldots$$
$$\texttt{let } xs' : [i][j] = xs[0{:}i, 0{:}j]$$

Directly lowering this code to an imperative context would either perform a deep copy of xs to compute xs' or modify all references to xs' to instead refer to the corresponding offset of xs. In both cases we lose information about the relation between xs and xs', which will make it harder to later analyze and reason about. Instead, we wish to stay in the functional context but add some additional memory-related information, by transforming our code into FUNMEM:

$$\texttt{let } xs_{mem} : \texttt{mem} = \texttt{alloc } (nm)$$
$$\texttt{let } xs : [n][m]@xs_{mem} \to L = \texttt{kernel } \ldots$$
$$\texttt{let } xs' : [i][j]@xs_{mem} \to \mathit{slice}(L, [0{:}i, 0{:}j]) = xs[0{:}i, 0{:}j]$$

We have added an additional statement, explicitly allocating the necessary memory for xs: xs_{mem}. In addition, we have associated with each array information about which memory allocation it belongs to, and how the values of the array are laid out in that buffer. For instance, xs is an array of size $n \times m$ which resides in xs_{mem} and is indexed using the index function L. Similarly, xs' is an $i \times j$ array that also resides in xs_{mem} with an appropriately sliced index function (see the definition of *slice* in section 3.4).

This way, we have kept the high-level relationship between xs and xs', while still allowing the compiler to reason about memory. The memory information that we add is *non-semantic*, meaning that if it is removed, the semantics of the program stays the same. As a result, programs in FUNMEM have two dynamic semantics we can use to evaluate them: A memory-agnostic value-based semantics that is equivalent to the one for FUN and a heap-based semantics that uses the memory annotations. A FUNMEM program is only valid if the two semantics agree on the result. Therefore, any memory optimizations introduced by the compiler must carefully preserve the semantics of the program.

We show the usefulness of our IR by presenting the full static and dynamic semantics for both FUN, FUNMEM and IMP, the latter language being a simple

[1]For concision, we are leaving out the element types of arrays in all our examples, so $[n][m]$ should be read "an $n \times m$ array of integers."

imperative target-language for our compilation, as well as simple transformations between the IRs. Additionally, we show an example of an optimization in FunMem called *memory expansion*, which hoists allocations out of parallel kernels in order to allow efficient execution on GPUs. Chapters 5 and 6 will show even further optimizations that rely on FunMem.

This chapter is structured as follows: We start with a brief discussion of the background for our work, as well as examples of related work. Then we introduce the Fun language, a simple functional IR without memory annotations. Afterwards we introduce FunMem, an extension of Fun with memory annotations and allocations, and we show how to transform a Fun program to FunMem. We then showcase the memory expansion optimization which uses FunMem. Finally, we introduce the third IR, Imp, which is used to illustrate how FunMem can be efficiently transformed into an imperative IR.

The contributions of this chapter are:

1. A formal specification of the static and dynamic semantics of the FunMem intermediate representation.
2. Procedures for translating Fun programs to FunMem and FunMem programs to Imp while preserving semantics.
3. An FunMem implementation of the memory expansion optimization for GPU execution of nested parallel kernels.

The work presented here has previously been published at SC22 [Mun+22] or will be published at IFL 2022 [MOHss].

4.2 Background and Related Work

Let's look at an expanded example of the one from the previous section:

$$
\begin{array}{l}
\texttt{let } xs : [n][m] = \texttt{kernel } \ldots \\
\texttt{let } xs' : [i][j] = xs[0{:}i, 0{:}j] \\
\ldots\ xs'[k,l]\ \ldots
\end{array}
$$

We still compute xs and xs' as before, but now we also have a use of xs' at a later point. The naive way of transforming our program into an imperative form and introducing memory is to manifest all intermediate arrays, meaning that xs' gets its own allocation:

$$
\begin{array}{l}
\texttt{var } xs_{mem} \texttt{ : mem} \\
xs_{mem} \leftarrow \texttt{alloc } (nm) \\
\texttt{kernel } \ldots \text{ -- Some computation that populates } xs_{mem} \\
\texttt{var } xs'_{mem} \texttt{ : mem} \\
xs'_{mem} \leftarrow \texttt{alloc } (ij) \\
\texttt{kernel } \ldots \text{ -- Copy values from } xs_{mem} \text{ to } xs'_{mem} \\
\ldots\ xs'_{mem}[kj + l]\ \ldots
\end{array}
$$

But of course, that introduces unnecessary overhead that wouldn't be present if we were writing our code directly in an imperative language: The reference to xs' should really be using the memory of xs. What we would like is something like the following, where q is the corresponding location of $xs'[k, l]$ inside xs_{mem}:

```
var xs_mem : mem
xs_mem ← alloc (nm)
kernel ... -- Some computation that populates xs_mem
... xs_mem[q] ...
```

This way get the desired reuse of the memory of xs, but we also lose information and expressibility in the process. For instance, if the $xs_{mem}[q]$ expression takes place inside a parallel loop and we want to make sure that the accesses are coalesced, we will need to change how xs_{mem} is laid out in memory, but it might not always be clear how that changes q or how other accesses to xs_{mem} are affected and how to change them accordingly. In short, we lose information about the original relation between xs' and xs.

Other approaches to introducing and handling memory in functional language IRs include destination-passing style [Sha+17], push and pull arrays [CSS12] as well as region inference [TB98].

Destination-passing style is used to efficiently allocate and deallocate memory by way of requiring function callers to explicitly pass in the memory for the function result, eliminating unnecessary copying of array results in higher-order functions. However, it does not solve the case of imperfectly nested maps, which still requires copying overheads.

Push and pull arrays use function composition to model array fusion, enabling compilers to sometimes use registers instead of expensive global memory. These however do not apply if fusion introduces redundant computation or if the compositions cannot be fused.

Region inference clusters together objects that share a lifetime, such that they can all be freed at the same time without garbage collection. This requires frequent allocation at the inner level which is very inefficient on GPUs.

In most cases, memory is introduced in functional IRs as part of a general lowering of the language into a more imperative style, as in the example above. This precludes further analysis of the memory usage at the level of the functional IR, making some optimizations harder, if not impossible, to write. The major exception is region inference, which does extend its functional IR with a non-semantic notion of memory, but it does so only to reason about the lifetime of objects in order to manage allocations and deallocations efficiently.

τ	::=		**Types**
	|	`int`	integer
	|	$[x]\cdots[x]$	array
v	::=		**Values**
	|	k	integer
	|	$[v,\ldots,v]$	array
p	::=	$(x:\tau)$	**Binding**
s	::=	`let` $p\cdots p = e$	**Statement**
b	::=		**Body**
	|	$s\ b$	statement
	|	`in` $(x,\cdots,x)$	result
se	::=		**Scalar expression**
	|	k	integer const
	|	x	variable
	|	$se \oplus se$	operator
e	::=		**Expressions**
	|	se	scalar expression
	|	$x[x,\ldots,x]$	index
	|	$x[x:x,\ldots,x:x]$	slice
	|	`transpose` x	transpose
	|	`if` x `then` b `else` b	conditional
	|	`kernel` $x \leq y$ `do` b	parallel loop
r	::=	$[k]\cdots[k]$	**Program result type**

Figure 4.1: Syntactic objects for FUN.

4.3 The FUN Language

We now introduce the FUN language. It is a pure, functional and size-dependently typed language with monomorphic types and parallelism, and corresponds to a subset of the functional IR used in Futhark. There are no support for functions or sequential loops in FUN, so it is insufficient to express any real programs, but it will serve to describe our technique.

Figure 4.1 shows the grammar for FUN. We assume a denumerably infinite set of program variables, ranged over x, y, z and use superscripts and subscripts to distinguish distinct variables. We also write $\overline{\alpha}$ to indicate a sequence of

αs, where α is a syntactical metavariable. We write $FV(\alpha)$ for the set of free variables in α.

Fun has two types: integers and arrays, the latter of which has explicit sizes for each dimension. For concision we have left out the base type of arrays, but it can be assumed to be `int`. A value can be an integer literal or an array literal. Statements consist of a number of patterns and an expression, binding the result of the expression to the names in the patterns. A body consists of a sequence of statements terminated by a result. An expression can be either a scalar expression, an array operation (indexing, slicing, transposing), a traditional conditional expression using `if` or a parallel loop using `kernel`. Slices have the form [offset : num_elems, ...], meaning that they do not, purely for simplicity of the language, support strides. Scalar operations consist of integer constants, variables and operators, where we assume the traditional arithmetic operations. `kernel` expressions consist of a thread-identifier and an upper bound as well as a body. In the expression `kernel` $x \leq y$ `do` b, y iterations of the loop body are executed in parallel, with x bound to the corresponding value between 1 and y in each iteration. The result of each iteration is written to the corresponding position in the result array. For example, the following statement computes an array of size y, containing the numbers from 2 to $y + 1$ and binds the result to xs:

$$
\begin{array}{l}
\texttt{let } xs : [y] = \texttt{kernel } x \leq y \texttt{ do} \\
\quad \texttt{let } z : \texttt{int} = x + 1 \\
\quad \texttt{in } (z)
\end{array}
$$

For simplicity of exposition, `kernel`-bodies can only return one value, but the rules for static and dynamic semantics could easily be expanded to accommodate multiple return values.

We also show the syntax for a program result type, which is used in the static semantics of Fun, discussed in section 4.3.1.

For convenience, we define the helper-functions `iota` and `copy`[2]. `iota` x computes an array of values from 1 to x, while `copy` x produces a copy of x:

$$
\begin{array}{l}
\texttt{iota } x \equiv \texttt{kernel } x_i \leq x \texttt{ do in } (x_i) \\
\texttt{copy } x \equiv \texttt{kernel } y_1 \leq z_1 \ \cdots \ \texttt{kernel } (y_n \leq z_n) \texttt{ do} \\
\qquad\qquad \texttt{in } (x[y_1, \ldots, y_n])
\end{array}
$$

Listing 4.1 shows an example of Fun statement, where $x_{size}, y_{size}, z_{cond}$ are assumed to be free. Note how the branches of the `if`-expression return not just the array value, but also the size needed in (the type of) the pattern of the `if` binding itself.

$\boxed{\Gamma \vdash se : \texttt{int}}$

$$\frac{}{\Gamma \vdash k : \texttt{int}}\ [\text{T-Const}] \qquad \frac{(x : \texttt{int}) \in \Gamma}{\Gamma \vdash x : \texttt{int}}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash se_1 : \texttt{int} \quad \Gamma \vdash se_2 : \texttt{int}}{\Gamma \vdash se_1 \oplus se_2 : \texttt{int}}\ [\text{T-Op}]$$

$\boxed{\Gamma \vdash s}$

$$\frac{\Gamma \vdash se : \texttt{int}}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = se}\ [\text{T-Scalar}]$$

$$\frac{(y : [z_1, \ldots, z_n]) \in \Gamma \quad \forall_1^n i.(y_i : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = y[y_1, \ldots, y_n]}\ [\text{T-Index}]$$

$$\frac{(x : \texttt{int}) \in \Gamma \quad \Gamma \vdash \overline{p} \leftarrow b_1 \quad \Gamma \vdash \overline{p} \leftarrow b_2 \quad \forall y \in FV(\overline{p}).(y : \texttt{int}) \in \Gamma, \overline{p}}{\Gamma \vdash \texttt{let}\ \overline{p} = \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_2}$$

[T-If]

$$\frac{(x_{arr} : [x_1][x_2]) \in \Gamma}{\Gamma \vdash \texttt{let}\ (y_{arr} : [x_2][x_1]) = \texttt{transpose}\ x_{arr}}\ [\text{T-Transpose}]$$

$$\frac{\forall_1^{2n} i.(y_i : int) \in \Gamma \qquad (x_{arr} : [x_1] \cdots [x_n]) \in \Gamma}{\Gamma \vdash \texttt{let}\ (z_{arr} : [y_2] \cdots [y_{2n}]) = x_{arr}[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}]}\ [\text{T-Slice}]$$

$$\frac{(y : int) \in \Gamma \qquad \Gamma, (x : \texttt{int}) \vdash (z : \tau) \leftarrow b \qquad \forall y' \in FV(\tau).(y' : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (z : [y]\tau) = \texttt{kernel}\ x \leq y\ \texttt{do}\ b}$$

[T-Kernel]

$\boxed{\Gamma \vdash \overline{p} \leftarrow b}$

$$\frac{\begin{array}{c} S = \{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\} \\ (x_1 : S(\tau_1)) \in \Gamma \quad \cdots \quad (x_n : S(\tau_n)) \in \Gamma \end{array}}{\Gamma \vdash (y_1 : \tau_1) \cdots (y_n : \tau_n) \leftarrow \texttt{in}\ (x_1, \cdots, x_n)}\ [\text{T-Result}]$$

$$\frac{\Gamma \vdash \texttt{let}\ \overline{p}^s = e \quad \Gamma, \overline{p}^s \vdash \overline{p} \leftarrow b}{\Gamma \vdash \overline{p} \leftarrow \texttt{let}\ \overline{p}^s = e\ b}\ [\text{T-Stm}]$$

Figure 4.2: FUN type rules.

```
let (z_size : int) (z_arr : [z_size]) =
  if z_cond then
    let (x : [x_size]) = iota x_size
    in (x_size, x)
  else
    let (x : [y_size]) = iota y_size
    in (y_size, y)
```

Listing 4.1: Example of FUN statement.

$$\boxed{\vdash b : (r, \ldots, r)}$$

$$\frac{\bullet \vdash (x_1 : [k_{1,1}] \cdots [k_{1,m_1}]) \cdots (x_n : [k_{n,1}] \cdots [k_{n,m_n}]) \leftarrow b}{\vdash b : ([k_{1,1}] \cdots [k_{1,m_1}], \ldots, [k_{n,1}] \cdots [k_{n,m_n}])} \text{ [T-PROG]}$$

Figure 4.2: Continued

4.3.1 Static Semantics

Next we introduce the static semantics, or type rules, for FUN, shown in fig. 4.2. The rules are pretty straightforward and consist of four judgments. The first judgment, $\Gamma \vdash se : \texttt{int}$ states that a particular scalar expression se is well-typed within the environment Γ and has the type `int`. The judgment $\Gamma \vdash s$ states that in the context Γ, the statement s is well-typed. Note that in the T-IF rule, we use $FV(\overline{p})$ to mean the free variables in the types of the patterns, specifically array sizes, not the names bound in the patterns themselves. In other words, the T-IF rule requires all array sizes occurring in types in $\overline{p}$ to either be bound by Γ or by another pattern in $\overline{p}$. The judgment $\Gamma \vdash \overline{p} \leftarrow b$ states that the body b is well-typed in the context of Γ and the result can be bound to the pattern $\overline{p}$. T-RESULT uses a substitution S to require that any arrays returned either already have their sizes bound in the surrounding environment or have their sizes returned as well. Finally, the judgment $\vdash b : (r, \ldots, r)$ states that b is a well-typed program returning arrays with constant-sized dimensions. Here, and for the rest of the thesis, we will use $\bullet$ to indicate an empty context and empty sequences in general.

4.3.2 Operational Semantics

We continue by introducing the operational semantics for the FUN language. We start with some helper functions defined in fig. 4.3. *index* is used to index

[2]We have taken some liberty with the notation for `copy`, but the idea is to have appropriately nested `kernel`-calls returning the corresponding values of the copied arrays.

$$
\begin{aligned}
&\mathit{index}(v, \bullet) && = v \\
&\mathit{index}([v_1, \ldots, v_m], k_1, \ldots, k_n) && = \mathit{index}(v_{k_1}, k_2, \ldots, k_n) \\
&\mathit{slc}(v, \bullet) && = v \\
&\mathit{slc}([v_1, \ldots, v_m], k_1 : k'_1, k_2 : k'_2 \ldots, k_n : k'_n) && = [\mathit{slc}(v_{k_1}, k_2 : k'_2, \ldots, k_n : k'_n), \\
& && \quad \vdots \\
& && \quad \mathit{slc}(v_{k_1+k'_1}, k_2 : k'_2, \ldots, k_n : k'_n)] \\
&\mathit{tr}([[v_{1,1}, \ldots, v_{1,n}], \ \ldots, [v_{m,1}, \ldots, v_{m,n}]]) && = [[v_{1,1}, \ldots, v_{m,1}], \\
& && \quad \vdots \\
& && \quad [v_{1,n}, \ldots, v_{m,n}]]
\end{aligned}
$$

Figure 4.3: Auxiliary functions for transforming values.

arrays according to a sequence of indexing constants. *slc* similarly slices an array and *tr* transpose.

Using these helper functions, we move on to the actual semantics, as shown in fig. 4.4. The semantics is a conventional value-based one, but note that we do not handle errors like out-of-bounds accesses and similar. Arrays are 1-indexed, primarily to make the notation concise. Here, and for the rest of the chapter, we use $E(\alpha)$ to denote substituting all variables in α with their corresponding values from the environment E, but also to evaluate arithmetic operations on the resulting constants, following conventional arithmetic rules.

The semantics consist of two judgments. The judgment $E \vdash e \Downarrow (v, \ldots, v)$ states that the expression e, when applied in the environment E, results in the values $(v, \ldots, v)$. The judgment $E \vdash b \downarrow (v, \ldots, v)$ states that the body b results in the values $(v, \ldots, v)$ when evaluated in the environment E.

4.4 FUN with Memory

Having introduced FUN, the base IR for our language, we now move on to FUNMEM, the memory-extended variant of FUN, which we can use for memory optimizations.

The syntax for FUNMEM can be seen in fig. 4.5. We reuse a lot of the constructs from FUN, but introduce a few new things: The `mem` type is the type of a memory allocation, which is the type of the result of the `alloc` expression. In addition to the sizes of their dimensions, arrays are now annotated with a variable indicating which memory allocation the array resides in, and an index function L, describing how individual values of the array are accessed. We will use LMADs, as described in chapter 3, as index functions. Additionally, for the operational semantics, heap labels and a heap abstraction is added. Listing 4.2 shows an example of a program in FUNMEM.

$\boxed{E \vdash e \Downarrow (v, \ldots, v)}$

$$\frac{E(x) \neq 0 \quad E \vdash b_1 \Downarrow (v_1, \ldots, v_n)}{E \vdash \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_2 \Downarrow (v_1, \ldots, v_n)}\ \text{[E-If-True]}$$

$$\frac{E(x) = 0 \quad E \vdash b_2 \Downarrow (v_1, \ldots, v_n)}{E \vdash \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_2 \Downarrow (v_1, \ldots, v_n)}\ \text{[E-If-False]}$$

$$\frac{v = \text{conventional evaluation of } se}{E \vdash se \Downarrow (v)}\ \text{[E-Scalar]}$$

$$\frac{v = \text{index}(E(x), E(y_1), \ldots, E(y_n))}{E \vdash x[y_1, \ldots, y_n] \Downarrow (v)}\ \text{[E-Index]}$$

$$\frac{v = \text{tr}(E(x))}{E \vdash \texttt{transpose}\ x \Downarrow (v) =}\ \text{[E-Transpose]}$$

$$\frac{v = slc(E(x), E(y_1){:}E(y_2), \ldots, E(y_{2n-1}){:}E(y_{2n}))}{E \vdash x[y_1{:}y_2, \ldots, y_{2n-1}{:}y_{2n}] \Downarrow (v)}\ \text{[E-Slice]}$$

$$\frac{\begin{array}{cc} & E, x \mapsto 1 \vdash b \Downarrow (v_1) \\ m = E(y) & \vdots \\ & E, x \mapsto m \vdash b \Downarrow (v_m) \end{array}}{E \vdash \texttt{kernel}\ x \leq y\ \texttt{do}\ b \Downarrow ([v_1, \ldots, v_m])}\ \text{[E-Kernel]}$$

$\boxed{E \vdash b \Downarrow (v, \ldots, v)}$

$$\frac{E \vdash e \Downarrow (v_1^e, \ldots, v_n^e) \quad E, x_1 \mapsto v_1^e, \ldots, x_n \mapsto v_n^e \vdash b \Downarrow (v_1^b, \ldots, v_m^b)}{E \vdash \texttt{let}\ (x_1 : \tau_1) \cdots (x_n : \tau_n) = e\ b \Downarrow (v_1^b, \ldots, v_m^b)}\ \text{[E-Let]}$$

$$\frac{E(x_i) = v_i \quad \text{for } 0 < i \leq n}{E \vdash \texttt{in}\ (x_1, \ldots, x_n) \Downarrow (v_1, \ldots, v_n)}\ \text{[E-In]}$$

Figure 4.4: Big-step operational semantics for Fun.

τ	::=		**Types**
	|	`int`	integer
	|	$[x]\cdots[x]@x \to L$	array
	|	`mem`	memory block
v	::=		**Values**
	|	k	integer
	|	ℓ	label
	|	(ℓ, L)	label and LMAD
	|	$[v, \ldots, v]$	array
e	::=		**Expressions**
	|	...	Any FUN expression
	|	`alloc` se	allocation
L	::=	$se + \{(x:x), \ldots, (x:x)\}$	**LMAD**
r	::=	$[x]\cdots[x]$	**Program result type**
H	::=	$\ell \mapsto [k, \ldots, k], H \quad \mid \quad \bullet$	**Heap**

Figure 4.5: Syntactic objects for FUNMEM. Most of the grammar is unchanged from FUN, but we require different information in types (τ) and we add an `alloc` expression.

```
let (z_size : int) (z_mem : mem) (z_arr : [z_size]@z_mem → R(z_size)) =
  if z_cond then
    let (x_mem : mem) = alloc x_size
    let (x : [x_size]@x_mem → R(x_size)) = iota x_size
    in (x_size, x_mem, x)
  else
    let (y_mem : mem) = alloc y_size
    let (x : [y_size]@y_mem → R(y_size)) = iota y_size
    in (y_size, y_mem, y)
```

Listing 4.2: The example from listing 4.1 translated to FUNMEM.

4.4.1 Static Semantics

The type rules for FunMem are shown in fig. 4.6. Though not depicted in the figures, we also reuse the $\Gamma \vdash se : \texttt{int}$ and $\Gamma \vdash \overline{p} \leftarrow b$ judgments from the Fun type rules in fig. 4.2. Furthermore, we use the definition of *slice* from fig. 3.6.

The judgment $\Gamma \vdash s$ states that the statement s is well-typed in the context Γ. It is important to notice that the memory of the original array is reused in the rules for transposition and slicing. Those operations are *change-of-layout* operations, and by using LMADs as index functions they become cost free ($O(1)$) operations at runtime because we are just changing how we index the values of the underlying memory buffer. Also note that the rule for kernels (T-Mem-Kernel) allows us to pick any index function for the resulting pattern, as long as the sizes of the dimensions fit. The rule for T-Mem-If states that all the types contained in the pattern must be well-typed in the context of $\Gamma, \overline{p}$.

The judgment $\Gamma \vdash L : [x_1] \cdots [x_n]$ states that in the environment Γ, the LMAD L is well-typed and describes an index function for an array of shape $[x_1] \cdots [x_n]$. Concretely, the T-Mem-LMAD rule states that all free variables in L must have type `int` in the context Γ, and that the index function matches the shape of the desired array.

The judgment $\Gamma \vdash \tau$ states that the type τ is well-typed in the context Γ. For array types, this means that the LMAD must be well-typed, and that the associated memory block must be bound within Γ.

The $\vdash b : (r, \ldots, r)$ judgment states that the body b is a well typed program returning constant-sized arrays. Note that we also require that all arrays return their memory allocations, that we enforce a particular ordering of the return values and that the index functions of the returned arrays are row-major. Forcing all array sizes to be constant and index functions to be row-major means that we don't need to return any additional data in order to determine how an array is laid out in memory.

The type rules for FunMem are unsound by design, meaning that well-typed programs can "go wrong". For instance, by not imposing any restrictions on which memory allocations are used for kernel-operations or how, we allow for imperative in-place updates. The type rules also allow race-conditions, by not verifying that different iterations of the same `kernel` expression read from and write to distinct memory locations. Of course, our goal is still that any FunMem program actually generated from Fun is sound, which we enforce through the concept of *validity*, described in section 4.4.3. By making the type rules more lenient, we can express various memory optimizations that would not otherwise be possible. Instead, we have opted to have the dynamic semantics include a check to verify the absence of data races, as described in section 4.4.2

$\boxed{\Gamma \vdash s}$

$$\frac{\Gamma \vdash se : \texttt{int}}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = se}\ [\text{T-Mem-Scalar}]$$

$$\frac{\Gamma \vdash se : \texttt{int}}{\Gamma \vdash \texttt{let}\ (x : \texttt{mem}) = \texttt{alloc}\ se}\ [\text{T-Mem-Alloc}]$$

$$\frac{(y : [z_1, \ldots, z_n]@y_{mem} \to L) \in \Gamma \quad \forall_1^n i.(y_i : \texttt{int}) \in \Gamma}{\Gamma \vdash \texttt{let}\ (x : \texttt{int}) = y[y_1, \ldots, y_n]}\ [\text{T-Mem-Index}]$$

$$\frac{(x : \texttt{int}) \in \Gamma \quad \Gamma \vdash \overline{p} \leftarrow b_1 \quad \Gamma \vdash \overline{p} \leftarrow b_2 \quad \forall (y : \tau) \in \overline{p}.\ \Gamma, \overline{p} \vdash \tau}{\Gamma \vdash \texttt{let}\ \overline{p} = \texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_2}\ [\text{T-Mem-If}]$$

$$\frac{(x_{arr} : [x_1][x_2]@x_{mem} \to L) \in \Gamma}{\Gamma \vdash \texttt{let}\ (y_{arr} : [x_2][x_1]@x_{mem} \to \mathit{transpose}(L)) = \texttt{transpose}\ x_{arr}}$$
$$[\text{T-Mem-Transpose}]$$

$$\frac{\begin{array}{c}(x_{arr} : [x_1] \cdots [x_n]@x_{mem} \to L_x) \in \Gamma \\ L_z = \mathit{slice}(L_x, [y_1 : y_2, \ldots, y_{2n-1} : y_{2n}])\end{array}}{\Gamma \vdash \texttt{let}\ (z_{arr} : [y_1] \cdots [y_{2n-1}]@x_{mem} \to L_z) = x_{arr}[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}]}$$
$$[\text{T-Mem-Slice}]$$

$$\frac{(y : int) \in \Gamma \qquad \Gamma, (x : \texttt{int}) \vdash (z : \tau) \leftarrow b \qquad \Gamma \vdash [y]\tau}{\Gamma \vdash \texttt{let}\ (z : [y]\tau) = \texttt{kernel}\ x \leq y\ \texttt{do}\ b}\ [\text{T-Mem-Kernel}]$$

$\boxed{\Gamma \vdash L : [x_1] \cdots [x_n]}$

$$\frac{L = se_o + \{(x_1 : y_1), \ldots, (x_n : y_n)\} \quad \forall z \in \text{FV}(L).(z : \texttt{int}) \in \Gamma}{\Gamma \vdash L : [x_1] \cdots [x_n]}$$
$$[\text{T-Mem-LMAD}]$$

$\boxed{\Gamma \vdash \tau}$

$$\frac{}{\Gamma \vdash \texttt{int}}\ [\text{T-Mem-Int}]$$

$$\frac{(x_{mem} : \texttt{mem}) \in \Gamma \quad \Gamma \vdash L_x : [x_1] \cdots [x_n]}{\Gamma \vdash [x_1] \cdots [x_n]@x_{mem} \to L_x}\ [\text{T-Mem-Arr}]$$

Figure 4.6: FunMem type rules.

$\boxed{\vdash b : (r, \ldots, r)}$

$$\frac{\begin{array}{c}\bullet \vdash p_1^{mem}\ p_1^{val} \cdots p_n^{mem}\ p_n^{val} \leftarrow b \\ \forall i.p_i^{mem} = (x_i^{mem} : \texttt{mem}) \\ \forall i.p_i^{val} = (x_i^{val} : [k_{i,1}] \cdots [k_{i,m_i}]@x_i^{mem} \to \mathcal{R}(k_{i,1}, \ldots, k_{i,m_i}))\end{array}}{\vdash b : ([k_{1,1}] \cdots [k_{1,m_1}], \ldots, [k_{n,1}] \cdots [k_{n,m_n}])}$$

[T-Mem-Prog]

Figure 4.6: Continued

$$\begin{array}{lll} \texttt{mem} & \Rightarrow_{\text{unmem}} & \texttt{int} \\ \overline{[x]}@x_{mem} \to L & \Rightarrow_{\text{unmem}} & \overline{[x]} \\ \texttt{alloc}\ se & \Rightarrow_{\text{unmem}} & 0 \end{array}$$

Figure 4.7: Turning a FunMem program into a Fun program. When these rules are applied at all possible locations to a FunMem program, the result will be a Fun program, although one that might redundantly shuffle extraneous information around, corresponding to the index functions and memory blocks that have been erased.

4.4.2 Operational Semantics

A FunMem program can be evaluated in one of two ways: Transforming it into a Fun program by essentially deleting the memory annotations and using the value-based semantics of Fun; or by using a heap-based semantics that mimics how FunMem would be implemented on actual physical hardware. A FunMem program is *valid* (a notion we will define with more rigidity in section 4.4.3) only if the two evaluation-methods produce the same result.

To evaluate a FunMem program under the value-based semantics, we first transform it into a Fun program. This is done by turning all values of type `mem` into `int` (they will have no semantic impact on the program), allocations into dummy integer literals and removing memory annotations from array types. In other words, we apply the rules in fig. 4.7 everywhere possible in the original program.

Alternatively, we can evaluate a FunMem program using the heap-based semantics shown in fig. 4.8. The semantics are a bit more complicated than the rules encountered so far, mostly due to the desire to keep a *trace* of memory accesses in order to guarantee *race-freedom*. As such, the $\boxed{\text{boxed}}$ parts serve to detect data races, but are not otherwise significant for the evaluation result. We use the definition of *slice* and *transpose* from fig. 3.6 as well as *racefree* from fig. 4.9 and *memcopy* from fig. 4.10. For space reasons we elide the rules for scalar expressions, as they are conventional.

The rules use a standard heap abstraction, denoted H, that maps labels ℓ

$$\boxed{E; H \vdash s \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}}$$

$$\frac{E(se) = m \quad \ell \; fresh \quad E' = E, y \mapsto \ell \quad H' = H, \ell \mapsto \overbrace{[0, \ldots, 0]}^{m}}{E; H \vdash \texttt{let}\ (y : \texttt{mem}) = \texttt{alloc}\ se \Downarrow E'; H'; \boxed{\langle \emptyset, \emptyset \rangle}}$$
[E-MEM-ALLOC]

$$\frac{\begin{array}{c} E(x) = (\ell_x, L_x) \\ E' = E, z \mapsto (\ell_x, \mathit{slice}(L_x, [E(y_1) : E(y_2), \ldots, E(y_{2n-1}) : E(y_{2n})])) \end{array}}{E; H \vdash \texttt{let}\ (z : \tau) = x[y_1 : y_2, \ldots, y_{2n-1} : y_{2n}] \Downarrow E'; H'; \boxed{\langle \emptyset, \emptyset \rangle}}$$
[E-MEM-SLICE]

$$\frac{\begin{array}{c} E(x) = (\ell_x, L_x) \\ E' = E, z \mapsto (\ell_x, \mathit{transpose}(L_x)) \end{array}}{E; H \vdash \texttt{let}\ (z : \tau) = \texttt{transpose}\ x \Downarrow E'; H'; \boxed{\langle \emptyset, \emptyset \rangle}}$$ [E-MEM-TRANSPOSE]

$$\frac{\begin{array}{c} E(x) = (\ell_x, L_x) \quad k = L_x(E(y_1), \ldots, E(y_n)) \quad E' = E, z \mapsto H[\ell_x, k] \\ \mathcal{R} = \{(\ell_x, k)\} \end{array}}{E; H \vdash \texttt{let}\ (z : \texttt{int}) = x[y_1, \ldots, y_n] \Downarrow E'; H'; \boxed{\langle \mathcal{R}, \emptyset \rangle}}$$
[E-MEM-INDEX]

$$\frac{\begin{array}{c} E, x \mapsto 1; H_0 \vdash b \Downarrow E_1; H_1; \boxed{\langle \mathcal{R}_1, \mathcal{W}_1 \rangle} \\ (H'_1, \boxed{\mathcal{R}'_1, \mathcal{W}'_1}) = \mathit{memcopy}(H_1, E_1(z_{res}), z_{mem}, \mathit{fix}(L_z, 1)) \\ \vdots \\ E, x \mapsto k; H'_{k-1} \vdash b \Downarrow E_k; H_k; \boxed{\langle \mathcal{R}_k, \mathcal{W}_k \rangle} \\ (H'_k, \boxed{\mathcal{R}'_k, \mathcal{W}'_k}) = \mathit{memcopy}(H_k, E_k(z_{res}), z_{mem}, \mathit{fix}(L_z, k)) \\ E(y) = k \quad \boxed{\mathit{racefree}(\mathcal{R}_1 \cup \mathcal{R}'_1, \ldots, \mathcal{R}_k \cup \mathcal{R}'_k, \mathcal{W}_1 \cup \mathcal{W}'_1, \ldots, \mathcal{W}_k \cup \mathcal{W}'_k)} \\ E' = E, z \mapsto (E(z_{mem}), E(L_z)) \\ \boxed{R' = \bigcup_{i=1}^{i \leq k} \mathcal{R}_i \cup \mathcal{R}'_i \quad W' = \bigcup_{i=1}^{i \leq k} \mathcal{W}_i \cup \mathcal{W}'_i} \end{array}}{E; H_0 \vdash \begin{array}{c} \texttt{let}\ (z : [z_1^d] \cdots [z_n^d] @ z_{mem} \to L_z) = \\ \texttt{kernel}\ x \leq y\ \texttt{do}\ \overline{s}\ \texttt{in}\ (z_{res}) \end{array} \Downarrow E'; H'_k; \boxed{\langle R', W' \rangle}}$$
[E-MEM-KERNEL]

Figure 4.8: Heap-based big-step operational semantics rules for FUNMEM.

$$\frac{\begin{array}{c}E(x) \neq 0 \quad E; H \vdash \overline{s} \Downarrow E_s; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle} \\ E' = E, x_1 \mapsto E_s(y_1), \ldots, x_n \mapsto E_s(y_n)\end{array}}{E; H \vdash \begin{array}{l}\texttt{let } (x_1 : \tau_1) \cdots (x_n : \tau_n) = \\ \quad \texttt{if } x \texttt{ then } \overline{s} \texttt{ in } (y_1, \ldots, y_n) \texttt{ else } b_2\end{array} \Downarrow E'; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle}} \text{ [E-Mem-If-T]}$$

$$\frac{\begin{array}{c}E(x) = 0 \quad E; H \vdash \overline{s} \Downarrow E_s; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle} \\ E' = E, x_1 \mapsto E_s(y_1), \ldots, x_n \mapsto E_s(y_n)\end{array}}{E; H \vdash \begin{array}{l}\texttt{let } (x_1 : \tau_1) \cdots (x_n : \tau_n) = \\ \quad \texttt{if } x \texttt{ then } b_1 \texttt{ else } \overline{s} \texttt{ in } (y_1, \ldots, y_n)\end{array} \Downarrow E'; H_s; \boxed{\langle \mathcal{R}_s, \mathcal{W}_s \rangle}} \text{ [E-Mem-If-F]}$$

$$\boxed{E; H \vdash \overline{s} \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}}$$

$$\frac{\begin{array}{c}E; H \vdash \texttt{let } \overline{p}_1 = e_1 \Downarrow E_1; H_1; \boxed{\langle \mathcal{R}_1, \mathcal{W}_1 \rangle} \\ \vdots \\ E_{n-1}; H_{n-1} \vdash \texttt{let } \overline{p}_n = e_n \Downarrow E_n; H_n; \boxed{\langle \mathcal{R}_n, \mathcal{W}_n \rangle}\end{array}}{E; H \vdash \texttt{let } \overline{p}_1 = e_1 \ \cdots \ \texttt{let } \overline{p}_n = e_n \Downarrow E_n; H_n; \boxed{\langle \bigcup_{i=1}^{i \leq n} \mathcal{R}_i, \bigcup_{i=1}^{i \leq n} \mathcal{W}_i \rangle}} \text{ [E-Stms]}$$

$$\boxed{\vdash b \Downarrow (v, \ldots, v); H}$$

$$\frac{\bullet; \bullet \vdash \overline{s} \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}}{\vdash \overline{s} \texttt{ in } (x_1, \ldots, x_n) \Downarrow (E(x_1), \ldots, E(x_n)); H} \text{ [E-Mem-Prog]}$$

Figure 4.8: Continued

to one-dimensional memory blocks, consisting of arrays of integers. We can look up the value at offset i in the memory block ℓ using $H[\ell, i]$. Similarly, we use $H[\ell, i] \mapsto k$ to construct a new heap identical to H but with index i in ℓ changed to k, modeling an in-place update.

The main judgment, $E; H \vdash s \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}$, states that in the value environment E and with the heap H, evaluating the statement s creates an extended value environment E', an updated heap H' and a trace of read and write locations, $\mathcal{R}$ and $\mathcal{W}$, represented as sets of (ℓ, L) pairs.

As mentioned, the trace has no semantic significance, but is used as a condition in E-Mem-Kernel to avoid data races by prohibiting locations written in one iteration of a parallel loop or kernel from being used in any way in any other iteration. The purpose is to allow implementations to concurrently execute different iterations of `kernel`. Note that locations are specific indices

$$\begin{array}{c} \mathit{racefree}(\mathcal{R}_1, \ldots, \mathcal{R}_k, \mathcal{W}_1, \ldots, \mathcal{W}_k) = \\ \forall_i (\mathcal{W}_i \cap \bigcup_{j \neq i} (\mathcal{R}_j \cup \mathcal{W}_j)) = \emptyset \end{array} \tag{4.1}$$

Figure 4.9: The definition of the *racefree* function.

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathit{memcopy}(H, k, \ell, L) = (H[\ell, L()] \mapsto k, \emptyset, \{(\ell, L())\})} \\ \multicolumn{3}{l}{\mathit{memcopy}(H, (\ell_{src}, L_{src}), \ell_{dst}, L_{dst}) = (H', \mathcal{R}, \mathcal{W})} \\ \text{where } H' & = & H[\ell_{dst}, i] \mapsto k_i \\ k_i & = & \begin{cases} H[\ell_{src}, j] & \exists\ j \in \mathit{dom}(L_{src}) \Rightarrow i = L_{dst}(j) \\ H[\ell_{dst}, i] & \text{otherwise} \end{cases} \\ \mathcal{R} & = & \{(\ell_{src}, p) \mid p \in \mathit{img}(L_{src})\} \\ \mathcal{W} & = & \{(\ell_{dst}, p) \mid p \in \mathit{img}(L_{dst})\} \end{array} \tag{4.2}$$

Figure 4.10: The definition of *memcopy*.

in a given memory block, not memory blocks themselves, so different iterations are allowed to access different locations within the same memory block.

E-Mem-Kernel is also the only place where the heap is actually modified, through the use of the *memcopy* function, seen in fig. 4.10. Simply put, we use *memcopy* to copy the results of each iteration of the `kernel` to the appropriate place in the result array. *memcopy* takes as arguments a heap H, a constant k or a pair of source label and index function, a destination label and destination index function, and returns an updated heap with values copied from the source to the destination as well as read and write sets. In short, if the result of a kernel body is a constant k, simply map the location given by the (zero-dimensional) destination LMAD to k. Otherwise, copy the values from the source array to the destination array element-wise, leaving any untouched values in place.

E-Mem-Alloc states that the `alloc` expression creates a new label and adds a freshly initialized array to the heap.

E-Mem-Slice and E-Mem-Transpose compute new arrays by slicing or transposing the LMAD of their inputs. Note that the memory block of the input is reused, meaning that the only cost of doing a slice is computing the new index function, which is done mostly at compile time. Only some additions and multiplications are left at runtime to compute the offset of a given index. Similarly, E-Mem-Index indexes an array by looking up the memory block and index function and computing the corresponding offset into the heap label. E-Mem-Index is the only place (outside of *memcopy*) where the read set is updated. E-Mem-If-T and E-Mem-If-F behave as expected.

Finally, the $E; H \vdash \overline{s} \Downarrow E; H; \boxed{\langle \mathcal{R}, \mathcal{W} \rangle}$ and $\vdash b \Downarrow (v, \ldots, v); H$ judgments describe how a sequence of statements and a program are evaluated.

$$
\begin{aligned}
&\mathit{nest}(k, \bullet) = k \\
&\mathit{nest}([v_1, \ldots, v_m], [k_1] \cdots [k_n]) = \\
&\quad [\mathit{nest}([v_1, \ldots, v_{k_n}], [k_1] \cdots [k_{n-1}]), \ldots, \mathit{nest}([v_{m-k_n}, \ldots, v_m], [k_1] \cdots [k_{n-1}])]
\end{aligned}
\tag{4.3}
$$

Figure 4.11: Definition of *nest*.

4.4.3 Validity

Having described the two ways of evaluating FunMem programs, we now move on to the concept of validity. The idea is to use the two evaluation-mechanisms to prove validity by requiring that a given program evaluates to equivalent results using both methods.

First, we define a helper function *nest* as shown in fig. 4.11, which produces a row-major view of the array in the given memory block. It takes a flat array and a shape as arguments and recursively computes the multi-dimensional array according to the shape. As an example, $\mathit{nest}([1, 2, 3, 4], [2][2])$ produces the array $[[1, 2], [3, 4]]$.

Next, we define validity:

Definition 4.4.1 (Validity)**.** Let b_{mem} be a FunMem program

$$\overline{s} \texttt{ in } (x_1^{mem}, x_1^{val}, \ldots, x_n^{mem}, x_n^{val})$$

and b be the corresponding Fun program given by

$$b_{mem} \Rightarrow_{unmem} b$$

Then b_{mem} is *valid* if

$$\vdash b_{mem} : ([k_{1,1}] \cdots [k_{1,m_1}], \ldots, [k_{n,1}] \cdots [k_{n,m_n}])$$

and

$$\vdash b_{mem} \Downarrow (\ell_1, (\ell_1, L_1), \ldots, \ell_n, (\ell_n, L_n)); H$$

and

$$\vdash b \Downarrow (0, v_1, \ldots, 0, v_n)$$

such that for all $1 \leq i \leq n$

$$\mathit{nest}(H(\ell_i), [k_{i,1}] \cdots [k_{i,m_i}]) = v_i.$$

Any process that creates a new FunMem program, including the initial translation from Fun, must take care to preserve validity.

Procedure 4.1: TRANSFORMPROGRAM(prg)

input : A well-typed FUN program $prg = \overline{s}$ `in` $(\overline{x})$.
output : A FUNMEM program corresponding to prg.

$\overline{s'}, \Gamma \longleftarrow$ TRANSFORMSTMS$(\overline{s}, \bullet)$;
$\overline{res} \longleftarrow \bullet$;
foreach x_i *in* $\overline{x}$ **do**
  $\overline{[z]}@mem \to L \longleftarrow \Gamma(x_i)$;
  $y_{mem}, y \longleftarrow fresh$;
  $s_{alloc} \longleftarrow$ `let` $(y_{mem} : \texttt{mem}) =$ `alloc` $(\prod z)$;
  $s_{lin} \longleftarrow$ `let` $(y : \overline{[z]}@y_{mem} \to \mathcal{R}(\overline{z})) =$ `copy` x_i;
  Append s_{alloc} s_{lin} to $\overline{s'}$;
  Append y_{mem}, y to $\overline{res}$;
return s' `in` (res);

4.5 FUN → FUNMEM

We now move on to describing how a FUN program is translated into FUNMEM. This process is handled by a collection of four procedures, shown in procs. 4.1 to 4.4.

TRANSFORMPROGRAM is the entry-point for the program transformation. It works by first calling TRANSFORMSTMS on the statements of the body of the given program and then performing a copy of the result values to ensure that they are in row-major form before returning them.

Procedure 4.2: TRANSFORMSTMS($\overline{s}$, Γ)

input : A sequence of FUN statements $\overline{s}$ and the corresponding type environment Γ.
output : The transformed FUNMEM statements $\overline{s'}$ with inserted memory and the corresponding type environment Γ'.

$\overline{s'} \longleftarrow \bullet$;
$\Gamma' \longleftarrow \Gamma$;
foreach s *in* $\overline{s}$ **do**
  $\overline{t} \longleftarrow$ TRANSFORMSTM(s, Γ');
  foreach `let` $\overline{p} = e$ *in* $\overline{t}$ **do**
    Append $\overline{p}$ to Γ';
  Append $\overline{t}$ to $\overline{s'}$;
return s', Γ';

TRANSFORMSTMS is a simple wrapper that just calls TRANSFORMSTM on each statement of a list of statements.

Procedure 4.3: TRANSFORMSTM(s,Γ)

input : A FUN statement $s : \texttt{let}\ \overline{p} = e$ and a FUNMEM type environment Γ.

output : FUNMEM statements $\overline{s'}$ corresponding to s with inserted memory annotations and memory allocations if necessary.

case $e \equiv se$ *or* $e \equiv x[y_1, \ldots, y_n]$ **do**
  return s;
case $\overline{p} \equiv (z : [z_1] \cdots [z_n])$ *and* $e \equiv x[y_1 : y_{n+1}, \ldots, y_n : y_{n+n}]$ **do**
  $[x_1] \cdots [x_n]@x_{mem} \to L_x \longleftarrow \Gamma(x)$;
  $L_y \longleftarrow \mathit{slice}(L_x, [y_1 : y_{n+1}, \ldots, y_n : y_{n+n}])$;
  return $\texttt{let}\ (z : [z_1] \cdots [z_n]@x_{mem} \to L_y) = e$;
case $\overline{p} \equiv (y : [n][m])$ *and* $e \equiv \texttt{transpose}\ x$ **do**
  $[m][n]@x_{mem} \to L \longleftarrow \Gamma(x)$;
  return $\texttt{let}\ (y : [n][m]@x_{mem} \to \mathit{transpose}(L)) = e$;
case $e \equiv \texttt{kernel}\ i \leq x\ \texttt{do}\ \overline{s'}\ \texttt{in}\ (x_{res})$ *and* $\overline{p} \equiv (y : [z_1] \cdots [z_n])$ **do**
  $\overline{s_{inner}}, \Gamma' \longleftarrow$ TRANSFORMSTMS($\overline{s'}, \Gamma$);
  $y_{mem} \longleftarrow \mathit{fresh}$;
  $s_{alloc} \longleftarrow \texttt{let}\ (y_{mem} : \texttt{mem}) = \texttt{alloc}\ (\prod_{i=1}^{n} z_i)$;
  $L_y \longleftarrow \mathcal{R}(z_1, \ldots, z_n)$;
  $p' \longleftarrow (y : [z_1] \cdots [z_n]@y_{mem} \to L_y)$;
  return $s_{alloc}\ \texttt{let}\ p' = \texttt{kernel}\ i \leq x\ \texttt{do}\ \overline{s_{inner}}\ \texttt{in}\ (x_{res})$;
case $e \equiv \texttt{if}\ c\ \texttt{then}\ \overline{s_{then}}\ \texttt{in}\ (\overline{x})\ \texttt{else}\ \overline{s_{else}}\ \texttt{in}\ (\overline{y})$ **do**
  $\overline{s'_{then}}, \Gamma_{then} \longleftarrow$ TRANSFORMSTMS($\overline{s_{then}}, \Gamma$);
  $\overline{s'_{else}}, \Gamma_{else} \longleftarrow$ TRANSFORMSTMS($\overline{s_{else}}, \Gamma$);
  $\overline{p_{res}}, \overline{x_{res}}, \overline{y_{res}} \longleftarrow \bullet, \bullet, \bullet$;
  foreach $x_i, y_i, (z_i : \tau_z)$ *in* $\overline{x}, \overline{y}, \overline{p}$ **do**
    $\tau_x \longleftarrow \Gamma_{then}(x_i)$;
    $\tau_y \longleftarrow \Gamma_{else}(y_i)$;
    $(\overline{s_x}; \overline{x_{supp}}) \longleftarrow$ SUPPORT(τ_x);
    $(\overline{s_y}; \overline{y_{supp}}) \longleftarrow$ SUPPORT(τ_y);
    Append $\overline{s_x}, \overline{s_y}$ to $\overline{s'_{then}}, \overline{s'_{else}}$ respectively;
    $S, \overline{p_i} \longleftarrow \bullet, \bullet$;
    foreach x' *in* $\overline{x_{supp}}$ **do**
      $x_{res} \longleftarrow \mathit{fresh}$;
      Append $x' \mapsto x_{res}$ to S;
      Append $(x_{res} : \texttt{int})$ to $\overline{p_i}$;
    Append $(z_i : S(\tau_x))$ to $\overline{p_i}$;
    Append $\overline{p_i}, \overline{x_{supp}}, \overline{y_{supp}}$ to $\overline{p_{res}}, \overline{x_{res}}, \overline{y_{res}}$ respectively
  return $\texttt{let}\ \overline{p_{res}} = \texttt{if}\ c\ \texttt{then}\ \overline{s'_{then}}\ \texttt{in}\ \overline{x_{res}}\ \texttt{else}\ \overline{s'_{else}}\ \texttt{in}\ \overline{y_{res}}$;

TransformStm, shown in proc. 4.3 is the main driver. It works by pattern matching on the expression and pattern of a statement, handling each kind of expression differently. Statements consisting of sub-expressions or array index expressions can be returned directly. When slicing or transposing an array x, we have to find the type and memory annotations for x in the environment first, after which the index function is transformed and a new statement is returned that reuses the memory of x.

`kernel`-statements are handled by first transforming the statements in the body of the `kernel`. Then, a new memory block is allocated with enough room for the values of all iterations and a suitable index function is computed. In this case, we always use a row-major index-function. Finally the `alloc`-statement and the updated `kernel`-statement are returned.

Handling of conditionals are slightly more complicated because we have to return the memory blocks and "supporting" information for the LMADs of all arrays that are returned. First we transform the statements of the two branches. Then, for each return value of the original statement, we look up the type in each branch and find all the supporting information using the Support function, shown in proc. 4.4. Given an array, Support returns a new statement binding the scalar expression in the offset to a variable, as well as the supporting information of the array: memory block and offset variable as well as spans and strides for each dimension. Then, for each of those supporting variables, we make up a new fresh variable name that we bind in the statement pattern. Finally, in order to make sure that the LMADs of returned arrays correctly use the returned existential values, we use a substitution. Note that we always return all strides, number of elements and memory blocks from each branch of a conditional. That may not be strictly necessary, but let's us keep the implementation simple. We leave it up to a later simplification pass to remove any redundant information, if desired.

Procedure 4.4: Support(τ)

input : A FunMem type τ
output : The supporting information of τ, and a statement binding the offset scalar expression to a variable, if there is one.

case $\tau \equiv$ `int` **do**
  return $(\bullet; \bullet)$;
case $\tau \equiv [x_1] \cdots [x_n]@x^{mem} \to se + \{(x_1 : x_{n+1}), \ldots, (x_n : x_{2n})\}$ **do**
  $y \longleftarrow fresh$;
  return (`let` $y = se; x^{mem}, y, x_1, \ldots, x_{2n}$);

All in all, these procedures allow us to systematically introduce memory in a Fun program resulting in a FunMem program. While providing a proof that this transformation is always valid is out of the scope of this thesis, we believe that this relatively simple procedure does indeed preserve validity.

4.6 Memory Expansion

Next, we will show how the FunMem IR can be used to perform powerful optimizations and code transformations in a simple way. The example we will use is that of *memory expansion*, which is critical for GPU code.

Because GPU kernels cannot efficiently allocate new memory for internal uses, it is common to instead pre-allocate enough memory for all the threads in a given kernel execution, and then have each thread use separate chunks of the allocated buffer. FunMem programs have no such limitation, but if we eventually do want to run our program on a GPU, we will need to get rid of any allocations inside `kernel`-expressions. We do so by hoisting them out of the `kernel` and expanding them accordingly, in a process called memory expansion. Assuming that a separate pass has been used to lift kernel allocations as much as possible, memory expansion can be performed on FunMem program using the simple procedure shown in proc. 4.5.

Procedure 4.5: MemoryExpand(prg)

input : A FunMem program prg where all memory annotations have been hoisted as much as possible.

output : A FunMem program where allocations at the top of `kernel` calls have been expanded out.

while *prg contains a statement*
$s \equiv$ `let` $\overline{p} =$ `kernel` $x \leq y$ `do let` $z =$ `alloc` $se\ b$
such that $x \notin FV(se)$ *and* z *is only used in patterns* **do**
  $z' \longleftarrow fresh$;
  $b' \longleftarrow b$ with all τ of the form
    $[x_1] \cdots [x_n]@z \rightarrow se' + \{(x_1 : x_{n+1}), \ldots, (x_n : x_{2n})\}$
  replaced with
    $[x_1] \cdots [x_n]@z' \rightarrow x \cdot se + se' + \{(x_1 : x_{n+1}), \ldots, (x_n : x_{2n})\}$
  Replace s with $s' =$
    `let` $z' =$ `alloc` $(se \cdot y)$
    `let` $\overline{p} =$ `kernel` $x \leq y$ `do` b'

MemoryExpand works by searching the given program for statements containing `kernel`-expressions, such that the first statement inside the `kernel`-body is an allocation of some variable z. We require the size of the allocation to not depend on the iterator value of the `kernel` and for z to only be used in patterns, i.e. z must not be returned. We then create a new variable z' which is used as the replacement allocation outside the body. We replace all references to z inside array types with z', and modify the associated LMADs with an offset expressed in terms of the iterator variable. The result is a simple procedure which expands memory out of kernels and modifies the index functions inside as necessary.

The programs resulting from the MemoryExpand procedure lay out the portions of memory belonging to different threads in separate contiguous chunks of memory. In other words, if each thread needs an array of n elements, the first thread will use the first n elements of the expanded memory buffer, and so on. This approach provides good spatial locality for CPU-based systems, since a given thread will access memory elements close to each other. For GPU systems on the other hand, this would be prohibitively expensive. Instead, we would rather have the accesses of different threads be interleaved, resulting in so-called *coalesced access*[3]. Thankfully, it is easy to produce an alternative version of MemoryExpand targeted at GPUs: We just have to use the following replacement LMAD instead of the one in the original formulation:

$$se' + x + \{(x_1 : yx_{n+1}), \ldots, (x_n : yx_{2n})\}$$

The result is that for threads with consecutive thread-IDs, elements with the same array index are laid out consecutively in memory. Strongly simplified, this means that when the threads are all accessing e.g. element 1 of their corresponding arrays, those elements are consecutive in memory so the GPU can fetch them in a single instruction, instead of having to do one fetch for each thread.

The MemoryExpand procedure shows how we can use the FunMem representation to perform high-level optimizations and transformations. Chapters 5 and 6 will show more complex examples of optimizations in FunMem.

4.7 An Imperative Target Language

We now introduce Imp, a simple imperative language with a parallel looping construct, which is used to illustrate how a FunMem program is translated into efficient imperative code.

Figure 4.12 shows the grammar for Imp, reusing the scalar expressions from Fun. The language has two types of variables, `int` and `mem`. Imp supports sequencing statements, declaration, assignment and allocations. It has support for reading and writing from one-dimensional memory buffers and it has branching and parallel loop constructs. The type rules for Imp are trivial and follow from the language, so we will not show them here. Listing 4.3 shows the FunMem program from listing 4.2 translated into Imp.

The dynamic semantics for Imp are shown in fig. 4.13. It is a fairly standard semantics that follows naturally from the language. It uses a heap abstraction H, like the one in Fun, mapping labels to memory blocks, and a value environment E, mapping variables to values. The judgement $H; E \vdash s \to H; E$ states that evaluating the statement s with the heap H and value environment E produces the new heap H' and environment E'.

[3]Consecutive SIMT threads concurrently executing load/store operation accessing consecutive locations in memory.

τ	::=		**Types**
	|	`int`	integer
	|	`mem`	memory
s	::=		**Statements**
	|	`skip`	no-op
	|	$s; s$	sequencing
	|	`var` x `:` τ	declaration
	|	$x \leftarrow se$	assignment
	|	$x \leftarrow$ `alloc` se	allocation
	|	$x \leftarrow x[se]$	read
	|	$x[se] \leftarrow se$	write
	|	`if` x `then` s `else` s `fi`	conditional
	|	`kernel` $x \leq se$ `do` s `done`	parallel loop

Figure 4.12: Grammar for Imp, a tiny imperative, structured, and statement-oriented language. Reuses the scalar expressions from the functional representation.

```
var z_size : int;
var z_mem : mem;
if z_cond then
  var x_mem : mem;
  x_mem ← alloc x_size;
  kernel i ≤ x_size do x_mem[i] ← i done;
  z_size ← x_size;
  z_mem ← x_mem
else
  var y_mem : mem;
  y_mem ← alloc y_size;
  kernel i ≤ y_size do y_mem[i] ← i done;
  z_size ← y_size;
  z_mem ← y_mem                                    fi
```

Listing 4.3: The example from listing 4.2 translated to Imp.

$\boxed{H; E \vdash s \to H; E}$

$$\frac{}{H; E \vdash \texttt{var}\ x\ :\ \texttt{int} \to H; E, x \mapsto 0}\ \text{[E-Imp-Dec-Int]}$$

$$\frac{}{H; E \vdash \texttt{var}\ x\ :\ \texttt{mem} \to H; E, x \mapsto \bot}\ \text{[E-Imp-Dec-Mem]}$$

$$\frac{}{H; E \vdash x \leftarrow se \to H; E, x \mapsto E(se)}\ \text{[E-Imp-Assign-Int]}$$

$$\frac{}{H; E \vdash x \leftarrow y \to H; E, x \mapsto E(y)}\ \text{[E-Imp-Assign-Mem]}$$

$$\frac{}{H; E \vdash x \leftarrow y[se] \to H; E, x \mapsto H[E(y), E(se)]}\ \text{[E-Imp-Read]}$$

$$\frac{}{H; E \vdash x[se_i] \leftarrow se_j \to H[E(x), E(se_i)] \mapsto E(se_j); E}\ \text{[E-Imp-Write]}$$

$$\frac{E(se) = m \quad \ell \text{ fresh} \quad H' = H, \ell \mapsto [\overbrace{0, \ldots, 0}^{m}]}{H; E \vdash x \leftarrow \texttt{alloc}\ se \to H'; E, x \mapsto \ell}\ \text{[E-Imp-Alloc]}$$

$$\frac{H_1; E_1 \vdash s_1 \to H_2; E_2 \quad H_2; E_2 \vdash s_2 \to H_3; E_3}{H_1; E_1 \vdash s_1; s_2 \to H_3; E_3}\ \text{[E-Imp-Seq]}$$

$$\frac{E(x) \neq 0 \quad H; E \vdash s_1 \to H', E'}{H; E \vdash \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \to H'; E'}\ \text{[E-Imp-If-True]}$$

$$\frac{E(x) = 0 \quad H; E \vdash s_2 \to H'; E'}{H; E \vdash \texttt{if}\ x\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \to H'; E'}\ \text{[E-Imp-If-False]}$$

$$\frac{}{H; E \vdash \texttt{skip} \to H; E}\ \text{[E-Imp-Skip]}$$

$$\frac{\begin{array}{c}E(se) = k \\ H; E, x \mapsto 1 \vdash s \to H'_1; E'_1 \quad \cdots \quad H'_{k-1}; E, x \mapsto k \vdash s \to H'_k; E'_k\end{array}}{H; E \vdash \texttt{kernel}\ x \leq se\ \texttt{do}\ s\ \texttt{done} \to H'_k; E}$$

[E-Imp-Kernel]

Figure 4.13: Big-step operational semantics rules for Imp. The heap structure is the same as for FunMem.

4.7.1 FunMem → Imp

Next, we describe how FunMem is translated into Imp. The main driver is FunMemToImp, as shown in proc. 4.6. FunMemToImp makes use of Copy which in turn can be seen in proc. 4.7.

FunMemToImp takes a statement s and an environment Γ and works by matching on the kind of expression used in s. For change-of-layout transformations such as slicing or transposing, we do not need to generate any code since these are free at runtime, so we return `skip`. For assignments of sub-expressions or allocations we declare the new variable and perform the assignment. To handle indexing of arrays, we first look up the memory block and index function of the array in question, and then we apply the index function to the given indices in order to find the correct offset into the corresponding memory block. For `if`-expressions we initialize the return variables outside the conditional and then we insert assignment statements at the end of each branch. Note that we have to filter out any array values in the returns and pattern, since Imp does not have array values. Finally, for `kernel`, we look up the type of the return value and insert a copy into the correct position in the pattern variable at the end of the body.

Translating an entire program is simply a matter of applying FunMemToImp to all statements in the program.

The Copy works as follows: If the value we're copying is an integer, simply index into the corresponding place in the destination memory and insert it there. Otherwise, if the result is an array, insert a `kernel`-nest of the appropriate depth and perform the copy in parallel.

Thus, we have described Imp and how we can translate FunMem into Imp in a straightforward manner. Imp corresponds closely to a standard imperative language (although with a parallel loop construct), so we provide an entire path from the functional IR, through FunMem to an imperative language. Again, we do not prove that the transformation from FunMem to Imp produces a valid program with equivalent semantic meaning, but we can use similar reasoning about validity as we did for the translation from Fun to FunMem.

4.8 Relation to IRs in Futhark

The languages presented in this chapter are simplified versions of internal languages used in the Futhark compiler. The actual implementation is much richer, containing more constructs such as sequential loops, more ways to construct fresh arrays than through `kernel` and more index transformations than just `transpose` and slicing. However, the core ideas of using LMADs to model the memory layout of arrays and the way that values are passed around through the program (particularly when returned from conditionals) are the same. Futhark also supports functions, which are implemented by adding the

Procedure 4.6: FunMemToImp(s, Γ)

input : A FunMem statement $s \equiv \texttt{let}\ \overline{p} = e$ and a FunMem type environment Γ containing all patterns in the *entire* program.

output : An Imp statement.

case $s \equiv \texttt{let}\ (x : \tau) = x[x_1 : x_2, \ldots, x_{n-1} : x_n]$ **do**
  return `skip`;

case $s \equiv \texttt{let}\ (x : \tau) = \texttt{transpose}\ x$ **do**
  return `skip`;

case $s \equiv \texttt{let}\ (x : \texttt{int}) = se$ **do**
  return $\texttt{var}\ x\ :\ \texttt{int}; x \leftarrow se$;

case $s \equiv \texttt{let}\ (x : \texttt{mem}) = \texttt{alloc}\ se$ **do**
  return $\texttt{var}\ x\ :\ \texttt{mem}; x \leftarrow \texttt{alloc}\ se$;

case $s \equiv \texttt{let}\ (x : \texttt{int}) = y[x_1, \ldots, x_n]$ **do**
  $\Gamma(y) \equiv \ldots @y^{mem} \rightarrow L^y$;
  return $\texttt{var}\ x\ :\ \texttt{int}; x \leftarrow y^{mem}[L^y(x_1, \ldots, x_n)]$;

case $s \equiv \texttt{let}\ \overline{p} = \texttt{if}\ y\ \texttt{then}\ b_1\ \texttt{else}\ b_2$ **do**
  $b_1 \equiv s_1^t \cdots s_n^t\ \texttt{in}\ \overline{x^t}$;
  $b_2 \equiv s_1^f \cdots s_n^f\ \texttt{in}\ \overline{x^f}$;
  $\forall (x_i : \tau_i) \in \overline{p} : \tau_i = \texttt{int} \vee \tau_i = \texttt{mem}.(x_i, \tau_i, x_i^t, x_i^f) \equiv (x_1', \tau_1', x_1^{t'}, x_1^{f'}), \ldots, (x_m', \tau_m', x_m^{t'}, x_m^{f'})$;
  $res \longleftarrow$
    $\texttt{var}\ x_1'\ :\ \tau_1'; \cdots; \texttt{var}\ x_m'\ :\ \tau_m'$;
    `if` y `then`
      FunMemToImp$(s_1^t); \cdots;$ FunMemToImp(s_n^t);
      $x_1' \leftarrow x_1^{t'}; \cdots; x_m' \leftarrow x_m^{t'}$
    `else`
      FunMemToImp$(s_1^f); \cdots;$ FunMemToImp(s_n^f);
      $x_1' \leftarrow x_1^{f'}; \cdots; x_m' \leftarrow x_m^{f'}$
    `fi`;
  return res;

case $s \equiv \texttt{let}\ p = \texttt{kernel}\ y_i \leq y_n\ \texttt{do}\ b$ **do**
  $b \equiv s_1 \cdots s_m\ \texttt{in}\ (x)$;
  $\Gamma(x) \equiv \tau$;
  return $\texttt{kernel}\ y_i \leq y_n\ \texttt{do}$
    FunMemToImp$(s_1); \cdots;$ FunMemToImp(s_m);
    Copy$(p, y_i, (x : \tau))$
  `done`;

Procedure 4.7: COPY(p_x, x^{idx}, p_y)

input : A pattern p_x, an index x^{idx}, and a pattern p_y.
output : IMP statement copying from p_y to $p_x[x^{idx}]$.

$p_x \equiv (x : [z_1] \cdots [z_n]@x^{mem} \to L_x)$;
case $p_y \equiv (y : \texttt{int})$ **do**
  return $x^{mem}[L_x(x^{idx})] \leftarrow y$;
case $p_y \equiv (y : [z_2] \cdots [z_n]@y^{mem} \to L_y)$ **do**
  return
    `kernel` $z_2^{idx} \leq z_2$ `do` $\cdots$ `kernel` $z_n^{idx} \leq z_n$ `do`
      $x^{mem}[L_x(x^{idx}, z_2^{idx}, \ldots, z_n^{idx})] \longleftarrow$
        $y^{mem}[L_y(z_2^{idx}, \ldots, z_n^{idx})]$
    `done ... done`;

necessary memory annotations and LMADs to function parameters and return values.

The Futhark compiler uses similar type rules to the ones presented here for each language in its internal representations, allowing it to type-check the results of each compiler-pass. As already discussed, this does not protect Futhark from all memory errors in FUNMEM, but it has caught many compiler bugs nonetheless. At the same time, FUNMEM enables compile-time memory optimizations that would be much harder to express without the high-level functional context that FUNMEM has.

Chapter 5

Memory Block Merging

The first major optimization is called "memory block merging". The goal is to identify memory blocks inside kernels that are no longer used but still live and use those instead of having to allocate new memory blocks. We show that register-allocation techniques can be applied to array allocations but also that it mostly matters for local memory.

5.1 Introduction and Motivation

Consider the following snippet of (extended) Fun code, where we assume that the computation of bs uses a but not as, and that as is no longer used after the computation of a:

$$\begin{aligned}&\texttt{let } as : [n] = \texttt{map } \ldots\\&\texttt{let } a : \texttt{int} = \texttt{reduce } (+)\ 0\ as\\&\texttt{let } bs : [m] = \texttt{map } \ldots\end{aligned}$$

The corresponding FunMem code would look like this:

$$\begin{aligned}&\texttt{let } as_{mem} : \texttt{mem} = \texttt{alloc } n\\&\texttt{let } as : [n]@as_{mem} \to \mathcal{R}(n) = \texttt{map } \ldots\\&\texttt{let } a : \texttt{int} = \texttt{reduce } (+)\ 0\ as\\&\texttt{let } bs_{mem} : \texttt{mem} = \texttt{alloc } m\\&\texttt{let } bs : [m]@bs_{mem} \to \mathcal{R}(m) = \texttt{map } \ldots\end{aligned}$$

We have two allocations, but since a_{mem} is never used after the computation of a, we could use a_{mem} to store bs instead of allocating a new memory block, but we have no way to express that in the source language (or Fun). In an imperative language, we could either directly reuse the allocation, or we could free as_{mem} after the computation of a and leave it up to the dynamic allocator to reuse the same memory for b_{mem}.

If our language uses a dynamic allocator, we can use a last use analysis to detect when a_{mem} is no longer used (after the computation of a) and then insert

a statement to free it. But because GPUs do not support dynamic memory allocation at all, it would only work outside kernels. Instead, we need to rely on static optimizations if we are to reduce the memory usage inside kernels[1]. To address such cases, we now describe the memory block merging analysis.

5.2 Related Work

Memory block merging is inspired by ordinary register allocation in compilers [Cha+81]. However, several additional constraints prevent us from using techniques such as linear scan [PS99]:

- Register allocation concerns itself with assigning values to a limited number of uniform registers, perhaps "spilling" any extraneous values to memory. In contrast, we are not in limited in the number of allocations we can make and there is no spilling. Rather, we take a program that *already has allocations in it*, and try to optimize those allocations.
- Our allocations are not uniform. Besides having different sizes, the elements of the arrays using each allocation can also have different sizes, and they can even reside in different "spaces"[2].

5.3 Intuition

Instead of using a linear scan algorithm, we are going to use graph coloring between the different allocations. The basic idea is to compute an interference graph between the memory blocks in the given program, and then use a standard greedy graph-coloring algorithm to merge non-interfering memory blocks.

An interference graph is a graph with edges between vertices that "interfere" with each other. In our case, memory blocks that are in use at the same time should interfere with each other. With the aid of information about the last use of each variable in a program, we can compute the interference graph using a top-down pass. Every time a use of an array is encountered, we insert edges between the associated memory block and all other memory blocks that are currently live, i.e. have been used at least once and have not reached their last use.

By inserting extra edges between allocations which reside in different spaces and which have different element sizes, we can adapt this general framework to our specific requirements.

We then use the interference graph to compute a coloring that distributes the arrays of the program onto memory blocks, insert the necessary allocations

[1] Therefore, memory block merging must also take place before memory expansion, described in section 4.6

[2] Global memory and local memory belong to different memory spaces on the GPU that cannot be intermixed.

and change the index functions in the program as necessary. Optimally coloring the graph is NP-complete [Kar72], so we will use a greedy non-optimal algorithm instead, which is also standard in register allocation [Cha+81].

5.4 An Example

Using the example from section 5.1, our optimization works as follows:

1. Compute the last use table for the program. For the example above, it is determined that as_{mem} is last used in the computation of a and that bs_{mem} is last used in the computation of bs.

2. Using the last use information to compute when memory-blocks are live, compute the interference graph of the memory blocks in the program. It is determined that as_{mem} and bs_{mem} do not interfere with each other because they are not live at the same time, have the same element-size and reside in the same memory space.

3. Using the greedy graph-coloring algorithm, we "color" the interference graph, assigning colors to all the memory blocks in the program and potentially merging memory blocks that do not overlap. The two memory blocks in our example are assigned the same color.

4. New allocation statements are inserted at the top of the kernel with fresh names. If necessary, a statement computing the maximum of the merged memory block sizes is inserted and the maximum is used as the allocation size. In this case, we will dynamically compute the maximum of n and m.

5. All index functions referring to the merged memory blocks are changed to refer to the new memory blocks.

The transformed code looks like this:

$$
\begin{array}{l}
\texttt{let } tmp : \texttt{int} = max(n, m) \\
\texttt{let } fresh_{mem} : \texttt{mem} = \texttt{alloc } tmp \\
\texttt{let } as_{mem} : \texttt{mem} = \texttt{alloc } n \\
\texttt{let } as : [n]@fresh_{mem} \to \mathcal{R}(n) = \texttt{map } \ldots \\
\texttt{let } a : \texttt{int} = \texttt{reduce } (+)\ 0\ as \\
\texttt{let } bs_{mem} : \texttt{mem} = \texttt{alloc } m \\
\texttt{let } bs : [m]@fresh_{mem} \to \mathcal{R}(m) = \texttt{map } \ldots
\end{array}
$$

Removing the now redundant allocations of as_{mem} and bs_{mem} is done by a later simplification pass.

5.5 Memory Block Merging in Futhark

Memory block merging has been fully implemented in the Futhark compiler, is part of the official release of Futhark and takes up around 800 lines of code. We have not found any significant increase in the performance of the generated code, but we have seen a reduction in memory usage in our OptionPricing benchmark[3] of 6%.

[3]Detailed in section 6.8.

Chapter 6

Short-Circuiting

We now introduce array short-circuiting, an optimization that aims to remove the overhead of parallelism safety guarantees in high-level languages by, among other things, allowing array updates to be performed in-place when deemed safe.

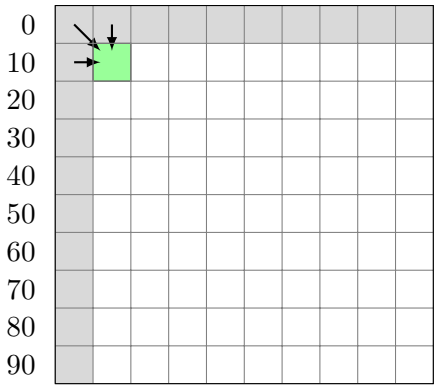
The array short-circuiting optimization has previously been presented at SC22 [Mun+22].

6.1 Motivation: NW

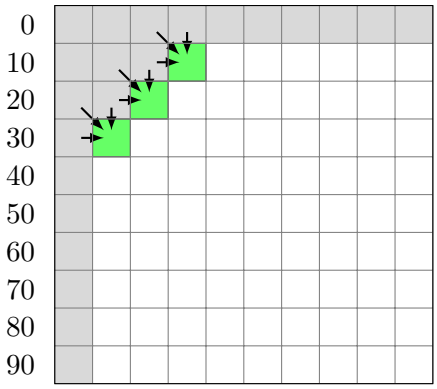
The Needleman-Wunsch benchmark, also known as NW, is an implementation of a dynamic programming algorithm for aligning protein sequences [Che+09]. The algorithm works by filling out the top and left perimeters of a two-dimensional array, and then computing the value of each cell depending on the three already computed neighbors, as seen in fig. 6.1a. Already computed values are marked in gray, while the values currently under evaluation are green. Based on the dependency pattern, we can parallelize the algorithm by computing all the cells on an antidiagonal in parallel, because none of the reads overlap with any of the writes, as shown in fig. 6.1b.

This naive parallelization of NW will incur lots of uncoalesced accesses to global memory on a GPU: Specifically three for each cell on the antidiagonal. Global memory is slow so we would rather use local memory, especially because the algorithm is largely memory-bound: The computation done for each cell is trivial. Some of the reads overlap, but moving those to local memory would only reduce the reads from global memory from $3n$ to $2n + 1$ (where n is the number of elements on the antidiagonal). Additionally, those reads would still be uncoalesced, and so would the writes at the end of each iteration of the main loop.

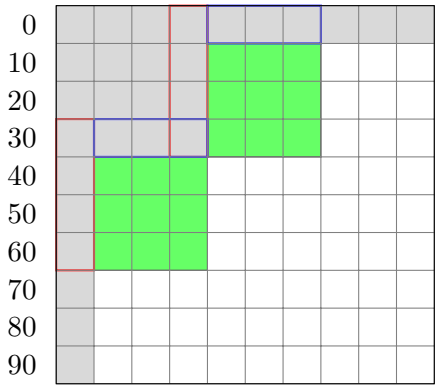
Instead, it is possible to block the algorithm, as shown in fig. 6.1c. The computation of each block only depends on the values immediately above and to the left of the block, shown in blue and red outlines respectively. The result



(a) Computing the value of a single cell.



(b) Computing values on an anti-diagonal in parallel.



(c) Computing a blocked anti-diagonal in parallel.

Figure 6.1: NW access patterns. Gray cells have already been computed.

is that each block consisting of $b \times b$ elements—in this case $b = 3$—reads $2b + 1$ elements from global memory, significantly increasing the performance of the algorithm, especially as b gets larger. In the example with $b = 3$, each block reads 7 elements from global memory using the blocked algorithm, compared to the $3 \cdot 3^2 = 27$ elements read with the naive parallelization. Even more importantly, both the reads of the top perimeter (outlined in blue in fig. 6.1c) and the writes of each block (in green) can be done in a coalesced fashion, greatly increasing the spatial locality of the resulting algorithm.

Using the in-place update notation described in section 2.2.2, the blocked version of the NW algorithm can be viewed as a map. If we denote by W the q blocks of $b \times b$ elements on a given antidiagonal of the original array, by R_1 the q slices of $b + 1$ elements shown in red in fig. 6.1c, by R_2 the q slices of b elements shown in blue in fig. 6.1c and by $\oplus$ some function computing the

block based on the two inputs, it can be expressed as:

$$\texttt{let } tmp = \texttt{map2} \ \oplus \ xs[R_1] \ xs[R_2]$$
$$\texttt{let } xs[W] = tmp$$

The goal of array short-circuiting is to compute tmp directly in the space of xs, eliminating the intermediate array.

Regular triplet notation does not allow us to express the slices necessary to formulate NW in this way, but LMADs-slices do: Assume that the original array is a one-dimensional array with n^2 elements (where $n = pb+1$, for some positive integer p and block size b) and i is the number of the blocked antidiagonal we are trying to compute ($i < p$). We can then use the notation from section 3.3 to describe the three slices above as:

$$W = n + ib + 1 + \{(i+1 : nb - b), (b : n), (b : 1)\}$$
$$R_1 = ib + \{(i+1 : nb - b), (b+1 : n)\}$$
$$R_2 = ib + 1 + \{(i+1 : nb - b), (b : 1)\}$$

The resulting arrays have the sizes:

$$xs[W] : [i+1][b][b]$$
$$xs[R_1] : [i+1][b+1]$$
$$xs[R_2] : [i+1][b]$$

As an example, in fig. 6.1c, we have $i = 1$, $b = 3$ and $n = 10$, which means that $W = 14 + \{(2:27), (3:10), (3:1)\}$ and $xs[W] : [2][3][3]$.

By expressing the slices this way, we can use a regular parallel `map` to compute the blocked anti-diagonal update forming the main loop of the program. However, due to the parallel-by-construction semantics of the `map`-SOAC and the freedom from data-races that it guarantees, the computation and write of the updated slice are separated. In other words, to statically ensure that none of the writes to xs overlaps with any of the reads from xs a temporary array is allocated containing the intermediate result. This protects the user from data-races, at the cost of incurring extra memory and copying overhead. As a result, when introducing memory to the code above, it is turned into the following:

$$\texttt{let } tmp_{mem} : \texttt{mem} = \texttt{alloc } ((i+1) * b * b)$$
$$\texttt{let } tmp : [i+1][b][b]@tmp_{mem} \rightarrow \mathcal{R}(i+1, b, b) =$$
$$\quad \texttt{map2} \ \oplus \ xs[R_1] \ xs[R_2]$$
$$\texttt{let } xs : [n^2]@xs_{mem} \rightarrow L_{xs} = xs \texttt{ with } [W] = tmp$$

Ideally, we would like to be able to construct *tmp* directly in the memory of xs, turning the code above into the following (where $slice_{\text{LMAD}}$ is taken from

fig. 3.6):

$$\begin{aligned}&\texttt{let } tmp : [i+1][b][b]@xs_{mem} \to slice_{\text{LMAD}}(L_{xs}, W) = \\ &\quad \texttt{map2 } \oplus\ xs[R_1]\ xs[R_2] \\ &\texttt{let } xs : [n^2]@xs_{mem} \to L_{xs} = xs\ \texttt{with}\ [W] = tmp\end{aligned}$$

While compiling this program to lower-level imperative code, the code generator can determine that the in-place update on the last line is a no-op and remove it altogether, eliminating the overhead of writing the intermediate array (*tmp*) to memory and reading from it again.

The safety guarantees in question produce a similar result when memory is introduced in concatenation-statements such as $\texttt{let } A = \texttt{concat } B\ C$[1]. The result is that B and C are computed in separate memory blocks before being copied into the result array. If B and C are never used again, we would like be able to save two allocations and some copying by constructing them directly in the memory space of A.

The in-place updates and constructions discussed above are easily written directly in imperative memory-oriented languages. In other words, the additional safety properties of functional array-languages prevent us from achieving the same performance as hand-written imperative code. To alleviate the situation, we propose the *array short-circuiting* optimization. The goal of this optimization is to identify instances of in-place updates or concatenations where it is safe to omit the intermediate buffer, such that the arrays are constructed directly in the desired location. This will allow our high-level functional programs to achieve the same level of performance as imperative code, without the safety hazards. The result of our optimization is that the NW update or the concatenation from above are turned into code without any unnecessary intermediate arrays, resulting in a speedup of 1.1×–2× on applicable benchmarks, as shown in section 6.8.

6.2 Overview

The short-circuiting optimization is a syntax-directed translation that works on the extended FunMem IR used in Futhark, but can be used in any IR that uses LMADs as array index functions. It starts by attempting to find a "short-circuiting point". Assuming the memory of x is last used in each statement, a short-circuiting point is one of the following:

1. $\texttt{let } ys = ys\ \texttt{with}\ [\ldots] = x$
2. $\texttt{let } ys = \texttt{concat } a\ x$
3. $\texttt{let } ys = \texttt{kernel } \ldots \texttt{ do } \ldots \texttt{ in } (x)$

[1] `concat` is a function that takes two arrays and concatenates them together, e.g. $\texttt{concat } [1,2]\ [3,4] = [1,2,3,4]$.

The `kernel` case arises because there is an implicit copy at the end of a kernel body, so we can regard it as attempting to put x directly in the space of ys during construction. Kernel short-circuiting like this is only valid if x is an array. In all cases, we call x the "source" and ys the "destination" of the short-circuiting attempt[2].

After finding a short-circuiting point, we first compute the projected index function, the index function that we would use for x if short-circuiting succeeds. Then, the optimization will attempt to prove that it is safe to construct the source directly in the space of the destination by traversing the program bottom-up from the short-circuiting point to the "array creation point" of the source and verifying the safety of the short-circuiting along the way.

The array creation point of an array x is a statement that creates a "new" array in memory, instead of simply changing the layout of an existing array. For a given array x and some z such that either $z = x$ or x aliases z, we support the following array creation points:

1. `let` $z =$ `kernel ...`
2. `let` $z =$ `iota ...`
3. `let` $z =$ `replicate ...`
4. `let` $z =$ `copy ...`

Note that the array creation point is distinct from the memory creation point, which is where the memory of z is allocated: They may be separated because of aggressive hoisting of memory allocations.

The goal of the bottom-up traversal is to verify the following properties:

1. The memory of the destination must be allocated before the creation of the source array.
2. Change-of-layout transformations both to and from the source are legal using the "projected" index functions as long as the projected index function can be constructed.
3. The free variables of the projected index function of the array are in scope at the array creation point, or can be brought into scope. In other words, the projected index function cannot rely on something we calculate later. Likewise for all aliases of the short-circuited array.
4. No writes to the source intersects with any later uses of the destination in between the array creation point and the short-circuiting point.

Upon reaching the array creation point and having verified these properties, the optimization deems it safe to compute the source directly in the destination memory space using the projected index function.

[2]Short-circuiting with a as the source in the `concat`-example is also valid.

6.3 Computing the Projected Index Function

Before validating the safety properties, the analysis needs to compute the projected index function of the source. When the short-circuiting point is an in-place update, the projected index function corresponds to the slice being updated. For instance, in the following example, the projected index function of x is $\mathit{slice}(L_{ys}, W)$:

$$\texttt{let } ys : \tau@ys_{mem} \to L_{ys} = ys \texttt{ with } [W] = x$$

When computing the projected index function for a `concat`-statement, we have to slice the index function of the destination accordingly. In the next example, if z is an array with n elements and x is an array with m elements, the projected index function of x is $\mathit{slice}(L_{ys}, [n:m])$:

$$\texttt{let } ys : \tau@ys_{mem} \to L_{ys} = \texttt{concat } z\ x$$

Lastly, for `kernel`-statements, each thread must write to a place in the destination according to its index. In the example below, the projected index function of x is $\mathit{fix}(L_{ys}, i)$ (where *fix*, which is defined in fig. 3.6, fixes the outermost dimension of the given LMAD):

$$\texttt{let } ys : [n]\tau@ys_{mem} \to L_{ys} = \texttt{kernel } i \leq n \texttt{ do } \ldots \texttt{ in } (x)$$

6.4 Safety Properties in Detail

We now describe in detail the properties outlined in section 6.2, which are needed to verify the safety of the short-circuiting transformation.

In most cases, the first safety property (that the array creation of the source must dominate the memory allocation of the destination) can be handled by a separate pass that attempts to lift all allocations as much as possible within a body, so we will not spend any more time on it here. The details of the rest of the safety properties are as follows:

6.4.1 Change-of-layout Transformations

The second safety property asserts that the necessary change-of-layout transformations on the projected index functions are possible. To illustrate this requirement, consider the following example, where a and b are not used at any later point (`transpose` is defined in fig. 3.6):

$$\texttt{let } a : [n][m]@a_{mem} \to L_a = \texttt{map } \ldots$$
$$\texttt{let } b : [m][n]@a_{mem} \to \mathit{transpose}(L_a) = \texttt{transpose } a$$
$$\texttt{let } ys : [p][m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = b$$

Short-circuiting starts by identifying the short-circuiting point on the third line, for which ys is the destination, b is the source and the projected index function is $\mathit{fix}(L_{ys}, i)$. Proceeding with the bottom-up analysis, we see that b is not the "original array", but rather that it is an alias of a. We therefore have to compute the projected index function of a by reverting the change-of-layout operation in question. Thankfully, transposition is easily reverted, in this case by computing $\mathit{transpose}(\mathit{fix}(L_{ys}, i))$, resulting in the following short-circuited code:

$$\begin{array}{l}
\texttt{let } a : [n][m]@ys_{mem} \to \mathit{transpose}(\mathit{fix}(L_{ys}, i)) = \texttt{map } \ldots \\
\texttt{let } b : [m][n]@ys_{mem} \to \mathit{fix}(L_{ys}, i) = \texttt{transpose } a \\
\texttt{let } ys : [p][m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = b
\end{array}$$

Unfortunately, some change-of-layout transformations cannot be reversed, as they are not information-preserving. Consider for instance the following:

$$\begin{array}{l}
\texttt{let } a : [n]@a_{mem} \to L_a = \texttt{map } \ldots \\
\texttt{let } b : [m]@a_{mem} \to \mathit{slice}(L_a, [x:y]) = a[x:y] \\
\texttt{let } ys : [n][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = b
\end{array}$$

The elements of the original array, a, do not fit into the space of ys that b is allowed to occupy, so we cannot short-circuit b into ys. Therefore, we only support reversing transpositions[3]. If our analysis encounters any non-reversible change-of-layout transformations, it will fail.

Just as the short-circuiting source may actually be the result of one or more change-of-layout operations on other arrays, other arrays may be the result of change-of-layout operations on the original array. For instance, consider the following example:

$$\begin{array}{l}
\texttt{let } a : [n][m]@a_{mem} \to L_a = \texttt{map } \ldots \\
\texttt{let } b : [m][n]@a_{mem} \to \mathit{transpose}(L_a) = \texttt{transpose } a \\
\texttt{let } c : [m]@a_{mem} \to \mathit{slice}(L_a, [x:y]) = a[x:y] \\
\texttt{let } ys : [p][m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = b
\end{array}$$

Because both b (our short-circuiting source) and c alias a, we have to record and change the index function of c in addition to a and b in order to short-circuit b into ys. Thankfully, this is trivial: Given the projected index function of a, we can always compute the index function of c by applying the appropriate change-of-layout function to the projected index function of a.

[3]It is an open question whether we could identify some cases in which the original array, e.g. a, is only used as the slice basis for the source array, e.g. b. In that case, it might be safe to shrink a such that it only contains the elements needed for b. Futhark's simplifier will handle some simple cases (by not computing the full a), but perhaps there is room for a more complex analysis.

After short-circuiting, the code above looks like this:

$$\texttt{let } a : [n][m]@ys_{mem} \to \mathit{transpose}(\mathit{fix}(L_{ys}, i)) = \texttt{map } \ldots$$
$$\texttt{let } b : [m][n]@ys_{mem} \to \mathit{fix}(L_{ys}, i) = \texttt{transpose } a$$
$$\texttt{let } c : [m]@a_{mem} \to \mathit{slice}(\mathit{transpose}(\mathit{fix}(L_{ys}, i)), [x:y]) = a[x:y]$$
$$\texttt{let } ys : [p][m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = b$$

6.4.2 Index-Function Projection Safety

When computing the projected index functions of short-circuited arrays and their aliases, we also have to ensure that the free variables in the new index function are actually in scope or can be computed when needed. For instance, consider the following example:

$$\texttt{let } ys_{mem} : \texttt{mem} = \texttt{alloc } m$$
$$\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = \ldots$$
$$\texttt{let } xs_{mem} : \texttt{mem} = \texttt{alloc } n$$
$$\texttt{let } xs : [n]@xs_{mem} \to L_{xs} = \ldots$$
$$\texttt{let } i : \texttt{int} = \ldots$$
$$\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = xs$$

In order to short-circuit xs into ys, we compute the projected index-function: $\mathit{fix}(L_{ys}, i)$. However, we cannot use this index function for xs, because i is not in scope when xs is created, it is only computed later. To handle these cases, we keep track of all the scalars used in the computation of the projected index functions and attempt to lift them above the array creation point of the short-circuiting source. In other words, the code above is turned into this:

$$\texttt{let } ys_{mem} : \texttt{mem} = \texttt{alloc } m$$
$$\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = \ldots$$
$$\texttt{let } xs_{mem} : \texttt{mem} = \texttt{alloc } n$$
$$\texttt{let } i : \texttt{int} = \ldots$$
$$\texttt{let } xs : [n]@xs_{mem} \to L_{xs} = \ldots$$
$$\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = xs$$

It is not always possible to lift the necessary scalar computations, for instance if the computation of the scalar in question depends on the short-circuiting source, as in the following example:

$$\texttt{let } ys_{mem} : \texttt{mem} = \texttt{alloc } m$$
$$\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = \ldots$$
$$\texttt{let } xs_{mem} : \texttt{mem} = \texttt{alloc } n$$
$$\texttt{let } xs : [n]@xs_{mem} \to L_{xs} = \ldots$$
$$\texttt{let } i : \texttt{int} = xs[0]$$
$$\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [i] = xs$$

In such cases, short-circuiting is not possible and will fail.

6.4.3 Non-Overlap

Finally, we have to ensure that short-circuiting does not introduce any data dependency errors. These can happen if a write to the source interferes with a later read from the destination, as in the following example:

$$
\begin{aligned}
&\texttt{let } ys_{mem} : \texttt{mem} = \texttt{alloc } m \\
&\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = \ldots \\
&\texttt{let } xs_{mem} : \texttt{mem} = \texttt{alloc } n \\
&\texttt{let } xs : [n]@xs_{mem} \to L_{xs} = \ldots \\
&\texttt{let } i : \texttt{int} = ys[0,0] \\
&\texttt{let } ys : [m][n]@ys_{mem} \to L_{ys} = ys \texttt{ with } [0] = xs
\end{aligned}
$$

In this case, short-circuiting xs into ys will cause i to have the wrong value, because the corresponding location in ys will have been overwritten during the creation of xs.

The simplest way to ensure safety from data dependency errors is to prevent the destination from being used in between the short-circuiting point and the array creation point of the source. Unfortunately, that is too restrictive for e.g. the NW example, where the computation of the blocked anti-diagonal depends on the values of adjacent cells in the destination. In the example above, if xs was instead placed in the second row of ys we would expect short-circuiting to succeed.

Instead of preventing all uses of the destination, short-circuiting will keep "access summaries" of the destination uses and source writes and use those to attempt to prove that short-circuiting is safe, even if the same array is being read and written to. An access summary is simply a set of LMADs, indicating what locations in the given memory block have been accessed.

The short-circuiting algorithm maintains two summaries of memory locations when trying to short-circuit some source x into some destination ys:

$\mathcal{U}_{ys}$**:** aggregates reads and writes of the destination array (ys, assumed to reside in ys_{mem}) and aliases thereof;

$\mathcal{W}_x$**:** aggregates writes to the source x and aliases thereof, using the projected index functions into ys_{mem}.

For each statement in our backwards pass towards the array creation point of x, we update each summary and verify at that point that none of the writes to the source overlap with previous uses of the destination. This guarantees that program semantics are preserved even if ys is accessed in the live range of x: Writes to x are not allowed to overlap with later reads or writes from ys. Checking non-overlap of the access summaries is done using the non-overlapping test from section 3.6 on the pair-wise LMADs from the two summaries.

-- Success if reached!
let x = scratch ...
-- $L_x^{wt} = slice(L_x^{new}, S_x^{wt})$
-- $\mathcal{W}_x = \{L_x^{wt}\}$
-- Fail here if: $\mathcal{U}_{ys} \cap L_x^{wt} \neq \emptyset$
let $x[S_x^{wt}] = \ldots$
-- $L_{ys}^{wt} = slice(L_{ys}, S_{ys}^{wt})$
-- $\mathcal{U}_{ys} = \mathcal{U}_{ys} \cup \{L_{ys}^{wt}\}$
let $ys[S_{ys}^{wt}] = \ldots$
-- $L_{ys}^{rd} = slice(L_{ys}, S_{ys}^{rd})$
-- $\mathcal{U}_{ys} = \{L_{ys}^{rd}\}$
let $\ldots = f\ (ys[S_{ys}^{rd}])$
-- $\mathcal{W}_x = \emptyset, \ \ \mathcal{U}_{ys} = \emptyset$
-- $L_x^{new} = slice(L_{ys}, S^{sc})$
let $ys[S^{sc}] = x$

Listing 6.1: Short-circuiting a straight line of code.

6.5 A Simple Example

Listing 6.1 shows an example of short-circuiting a straight line of code, and the resulting access summaries.

We start with the short-circuiting point on the last line: The goal is to short-circuit x into ys using the slice S^{sc}. We initialize the access summaries $\mathcal{W}_x$ and $\mathcal{U}_{ys}$ to the empty sets and compute the projected index function, L_x^{new}, for x.

On the next two lines, we encounter a read from and a write to the destination. In both cases, we compute the index function corresponding to access and add it to the access summary. Then we encounter a write to the source, meaning that we have to compute the corresponding access summary and add it to the write set. If the write to the source overlaps with any of the uses of the destination encountered so far, short-circuiting fails. Otherwise the analysis continues to the first line, the array creation point of x. If the analysis reaches this point, the short-circuiting analysis succeeds and we can update the index function of x accordingly.

6.6 Handling Recurrences

Constructs like loops and maps introduce additional complexity to our analysis. For instance, we have to check the access summaries of each iteration of a sequential loop against all the previous iterations. Likewise when handling a

parallel loop we have to check the accesses of each iteration against all other iterations.

6.6.1 Loops

To handle sequential loops, we first apply the short-circuiting analysis to the loop body using empty initial access summaries, verifying that short-circuiting is safe inside to body of a single iteration. Having collected and verified the access summaries of a single iteration denoted by the iteration index i, $\mathcal{U}_{ys}^{i}$ and $\mathcal{W}_{x}^{i}$, we then denote by $\mathcal{U}_{ys}^{>i}$ the access summary of all the destination uses in iterations after iteration i. Similarly, we denote by $\mathcal{U}_{ys}^{\text{total}}$ and $\mathcal{W}_{x}^{\text{total}}$ the total access summaries for all the iterations of a loop. The three aggregated access summaries are defined as follows and can be computed using the LMAD expansion procedure, EXPAND, described in section 3.5.2:

$$\mathcal{U}_{ys}^{>i} = \bigcup_{j=i+1}^{n-1} \mathcal{U}_{ys}^{j}$$
$$\mathcal{U}_{ys}^{\text{total}} = \bigcup_{j=0}^{n-1} \mathcal{U}_{ys}^{j}$$
$$\mathcal{W}_{x}^{\text{total}} = \bigcup_{j=0}^{n-1} \mathcal{W}_{x}^{j}$$

Assuming that $\mathcal{U}_{ys}$ is the use set of the destination after the loop, we can check inter-iteration overlap, as well as overlap of writes inside the loop with any previously encountered uses of the destination, using the following equation:

$$\mathcal{U}_{ys}^{>i} \cap \mathcal{W}_{x}^{i} = \emptyset \quad \wedge \quad \mathcal{U}_{ys} \cap \mathcal{W}_{x}^{\text{loop}} = \emptyset$$

Finally, to proceed with our analysis, we update the outside access summaries by adding the total access summary of the loop:

$$\mathcal{W}'_{x} = \mathcal{W}_{x} \cup \mathcal{W}_{x}^{\text{total}}$$
$$\mathcal{U}'_{ys} = \mathcal{U}_{ys} \cup \mathcal{U}_{ys}^{\text{total}}$$

6.6.2 Parallel Loops

Kernels are handled similarly to sequential loops, except for the fact that all the threads of a kernel can execute simultaneously. Therefore, instead of $\mathcal{U}_{ys}^{>i}$, we need to compute $\mathcal{U}_{ys}^{\neq i}$, which is defined by the following equation:

$$\mathcal{U}_{ys}^{\neq i} = \mathcal{U}_{ys}^{<i} \cup \mathcal{U}_{ys}^{>i} = \bigcup_{j=0}^{i-1} \mathcal{U}_{ys}^{j} \cup \bigcup_{j=i+1}^{n-1} \mathcal{U}_{ys}^{j}$$

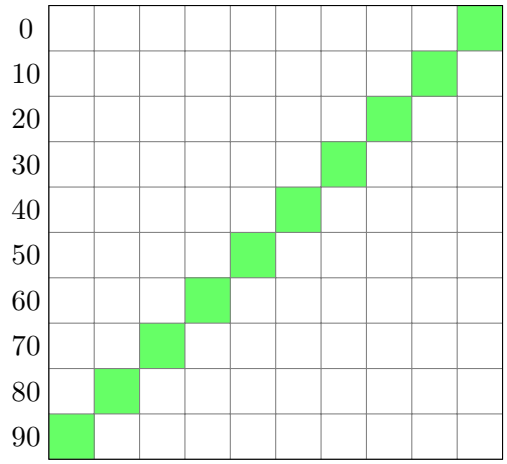


Figure 6.2: The updated anti-diagonal elements computed in listing 6.2.

We also have to take into consideration that each thread in a map ends with an implicit write to the result array. So if the result of the parallel loop is the source of our current short-circuiting attempt, x, with projected index function L_x and the thread ID i, then the body of a `kernel` expression ends with a source write defined by:

$$\mathcal{W}_x^{i,\text{init}} = \mathit{fix}(L_x, i)$$

Thus, when analyzing the body of the parallel loop, the access summary for writes to the source is initialized to $\mathcal{W}_x'^{i,\text{init}}$.

Like for sequential loops, if we assume that $\mathcal{U}_{ys}$ is the use set of the destination after the loop, inter-iteration overlap is checked by the following equation:

$$\mathcal{U}_{ys}^{\neq i} \cap \mathcal{W}_x^i = \emptyset \quad \wedge \quad \mathcal{U}_{ys} \cap \mathcal{W}_x^{\text{total}} = \emptyset$$

Likewise, to proceed with our analysis, we update the outside access summaries by adding the total access summary of the parallel loop:

$$\mathcal{W}_x' = \mathcal{W}_x \cup \mathcal{W}_x^{\text{total}}$$
$$\mathcal{U}_{ys}' = \mathcal{U}_{ys} \cup \mathcal{U}_{ys}^{\text{total}}$$

6.6.3 An Example with a Parallel Loop

Listing 6.2 shows a concrete example of short-circuiting code involving a `kernel`. By regarding ys as a two-dimensional array with 10×10 elements, the code in question increments the values of the anti-diagonal in parallel, following the pattern shown in fig. 6.2.

```
let ys_mem : mem = alloc 100
let ys : [100]@ys_mem → R(100) = ...
let ys' : [10]@ys_mem → 9 + {(10:9)} = ys[9 + {(10:9)}]
let x_mem : mem = alloc 10
-- U_ys = U_ys^total = {9 + {(10:9)}}
-- W_x = {9 + 9i, W_xs^total} = {9 + 9i, 9 + {(10:9)}}
-- Array-creation of x, analysis succeeds
let x : [10]@x_mem → R(10) =
  -- U_ys^{≠i} = ⋃_{j=0}^{j<i} U_ys^j ∪ ⋃_{j=i+1}^{j<10} U_ys^j
  --           = {9 + {(i:9)}, 9i + 18 + {(10 − i − 1:9)}}
  -- Verify that U_ys^{≠i} ∩ W_x^i = ∅
  kernel i ≤ 10 do
    -- U_ys^i = {9 + 9i}, W_x^i = {9 + 9i}
    let tmp : int = ys'[i] + 1
    -- U_ys^i = {}, W_x^i = {9 + 9i}
    in (tmp)
-- U_ys = {}, W_x = {}
-- Projected index function for x: 9 + {(10:9)}
let ys[9 + {(10:9)}] = x
```

Listing 6.2: Short-circuiting with a kernel.

Starting from the last line, we first find the short-circuit point: x is being used to update ys in-place. The projected index function of x is computed and the access summaries, $\mathcal{U}_{ys}$ and $\mathcal{W}_x$, are initialized.

Next, we encounter a `kernel`-expression, a parallel loop. To handle it, we must first process the body independently, computing the access summaries $\mathcal{U}^i_{ys}$ and $\mathcal{W}^i_x$. On the last line of the body we have the return statement which contains an implicit write to the source array x. Therefore, we must verify that the implicit write does not overlap with any previous uses of the destination. Since $\mathcal{U}^i_{ys}$ is empty, that follows trivially, and we update $\mathcal{W}^i_x$ accordingly.

On the next line of the kernel body we have a read from an array aliasing the destination (ys'), so the corresponding index is added to $\mathcal{U}^i_{ys}$. Note that, even though this read from the destination overlaps with the implicit write to the source at the end of the kernel, the write comes after the read so short-circuiting will not violate the original dependencies.

Once the entire body has been processed, we have to verify that the source writes in each iteration do not overlap with any destination uses in the other threads. We therefore compute $\mathcal{U}^{\neq i}_{ys}$ and attempt to prove that it is disjoint from $\mathcal{W}^i_x$ using the non-overlapping test from section 3.6. If successful, the analysis updates $\mathcal{U}_{ys}$ and $\mathcal{W}_x$ using the access summaries of the entire loop

$\mathcal{U}_{ys}^{total}$ and $\mathcal{W}_{x}^{total}$ before proceeding with the analysis.

Intuitively, it can be seen that $\mathcal{W}_x^i = \{9 + 9i\}$ is disjoint from $9 + \{(i:9)\}$ because the latter corresponds to the points $\{9, 18, \ldots, 9 + 9(i-1)\}$. Similarly, $\mathcal{W}^i - x$ is disjoint from $9i + 18 + \{(10 - i - 1:9)\}$ because the latter corresponds to the points $\{9i + 18, 9i + 27, \ldots, 90\}$.

The statement computing x is also the array creation point of the source of our current short-circuiting attempt. Thus, because the analysis has not encountered any dependency errors along the way, short-circuiting x into ys is deemed safe and the index function of x can be updated accordingly.

6.7 Implementation Details

Short-circuiting has been fully implemented as an automated pass in the Futhark compiler and takes up around 5000 lines of Haskell code.

Proving that access summaries do not overlap is the most complex part of the pass. While our heuristic from section 3.6.2 allow us to use simple inequalities to prove non-overlap of complex examples like NW and the LUD benchmark (which we will describe in section 6.8.2), the algebraic simplifier and solver within Futhark is not yet up to that task. We therefore have two versions of the short-circuiting pass implemented: One that uses our internal algebraic simplifier but cannot prove non-overlap in NW and LUD (though it does improve other benchmarks, and parts of LUD and NW) as well as one that offloads the algebraic simplification and solving of inequalities to a third-party SMT-solver (specifically, Z3 [MB08]), which can fully short-circuit NW and LUD. To keep the number of external dependencies in the official Futhark releases down, the first one is the one that is used in the current master-branch of Futhark, while the latter one lives in its own branch.

While having to use an external SMT-solver to fully optimize cases like NW might seem like a detriment to our implementation, it is important to note that the SMT-solver is not able to prove non-overlap of NW on its own. Indeed, when given the raw LMADs in order to try to prove them disjoint, Z3 will run for more than a week without giving an answer. Only by giving it the preprocessed inequalities arising from the heuristic described in section 3.6.2 is it able to produce any answer at all. We believe that implementing the necessary algebraic solver-procedures directly in the Futhark compiler is feasible, though outside the scope of this work[4].

Short-circuiting causes a compile time overhead of around 10% on most of our benchmarks. However, some benchmarks take even longer to compile with short-circuiting due to the complex LMAD that arise. For instance, NW compiles in 17 seconds using the external SMT-solver, compared to 1 second without.

[4]And probably not scientifically interesting. Lots of work exists concerning the implementation of algebraic simplification engines and solvers.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Speedup	Optimization Impact
A100	8192	9	0.99x	1.16x	1.17x
	16384	21	0.96x	1.19x	1.24x
	32768	58	1.04x	1.36x	1.31x
MI100	8192	15	0.71x	0.88x	1.24x
	16384	44	0.64x	0.78x	1.21x
	32768	325	1.01x	1.14x	1.13x

Table 6.1: NW performance. 1000 runs.

6.8 Experimental Results

To investigate the impact of the short-circuiting optimization, we have evaluated it on a collection of benchmarks. All benchmarks were run using NVIDIA A100 and AMD MI100 GPUs. To get accurate measurements, each benchmark is run enough times (as indicated by the header of each table) for the runtimes to settle at a stable level. We always discard the first run and then average the rest. All benchmarks are compared to a reference hand-written OpenCL implementation from a publicly available benchmark suite, and we show the relative performance of the Futhark implementation both with and without short-circuiting, as well as the impact of the optimization.

6.8.1 NW

We benchmark our NW implementation against the hand-written OpenCL implementation from the Rodinia benchmark suite [Che+09]. The results are shown in table 6.1, the number in the dataset-column indicates the size of the square matrix used as input.

For the largest dataset, our short-circuited Futhark-code is 36% faster than the reference implementation on the A100, with the optimization itself having an impact of 31%. The code generated by Futhark is very similar to the hand-written implementation, so the difference in performance is likely due to a bank conflict in the Rodinia code that happens when accessing shared memory. The Futhark version uses padding to avoid such conflicts. On the MI100, things are not quite as impressive, but for the largest dataset our implementation is 14% faster than the reference, with a short-circuiting impact of 13%.

6.8.2 LUD

The LUD benchmark performs lower-upper decomposition of a matrix. We compare our implementation with the LUD implementation from Rodinia, to which the code generated by Futhark is also very similar.

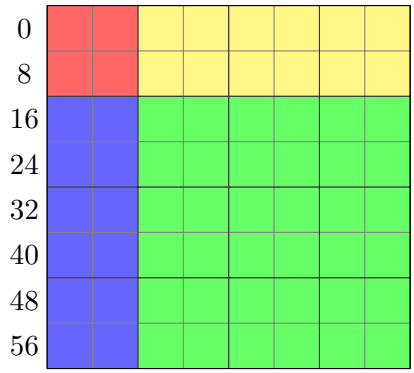


Figure 6.3: LUD access pattern.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Speedup	Optimization Impact
A100	8192	190	1.08x	1.34x	1.25x
	16384	1445	1.19x	1.53x	1.29x
	32768	11547	1.21x	1.60x	1.32x
MI100	8192	173	0.60x	0.72x	1.19x
	16384	1248	0.74x	0.98x	1.32x
	32768	10511	0.83x	1.14x	1.39x

Table 6.2: LUD performance. 10 runs.

At a high level, the algorithm blocks the input matrix and iterates along the diagonal axis of blocks, processing a successively smaller part of the matrix. Figure 6.3 shows an example of the access pattern with a block-size of two: The red block is computed first. The result is then used to compute the blue and yellow blocks, all three of which are used to compute the green blocks at the end. Then, the algorithm recursively processes the green blocks in a similar manner.

Table 6.2 shows the performance of our implementation compared to the reference implementation from Rodinia. Like for NW, the number in the dataset column indicates the size of the square matrix used as input. On the A100 we are 60% faster for the largest dataset with an impact from the optimization of 32%. On the MI100 we are 14% faster for the largest dataset with an optimization impact of 39%. The difference in performance is largely due to the fact that Futhark automatically inserts register-tiling in addition to the block-tiling expressed by the user.

6.8.3 Hotspot

The Hotspot benchmark models heat propagation in processors and consists of a big stencil. It takes as input a two-dimensional matrix and a number of

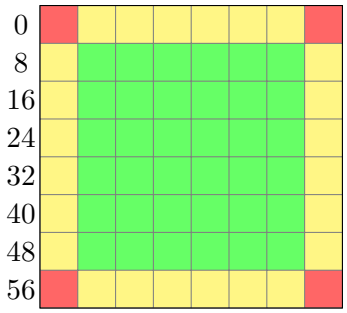


Figure 6.4: Hotspot access pattern.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Futhark	Optimization Impact
A100	8192	9	0.47x	0.84x	1.78x
	16384	29	0.46x	0.94x	2.04x
	32768	117	0.46x	0.94x	2.05x
MI100	8192	8	0.33x	0.64x	1.96x
	16384	34	0.35x	0.68x	1.97x
	32768	142	0.37x	0.73x	1.98x

Table 6.3: Hotspot performance. 10 runs.

iterations. In each iteration, the values of the elements of the two-dimensional matrix is updated based on its cardinal neighbors. To compute this in parallel, we first compute the values at the corners and edges of the matrix and then compute the inner values. The resulting access pattern can be seen in fig. 6.4, where the differently colored cells are computed in separate parallel operations.

By constructing the final matrix as the concatenation of those different parts, short-circuiting can make sure that the values are actually constructed in-place, leading to around 2x speedup compared to the un-short-circuited version, as seen in table 6.3 (again, the dataset number is the size of the matrix). In terms of absolute performance, the optimized Futhark version is close to that of the hand-written implementation from Rodinia on the A100, but a bit further away on the MI100.

6.8.4 LBM

The LBM benchmark is an implementation of the Lattice Boltzmann methods for simulating fluid dynamics. The main part of the code consists of a loop with a map inside, computing the rows of the output array. Our short-circuiting method allows Futhark to compute the intermediate results of that main map directly in the desired output space, removing extra copies from the resulting code.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Speedup	Optimization Impact
A100	short	29	0.84x	0.92x	1.09x
	long	860	0.86x	0.95x	1.10x
MI100	short	49	0.65x	1.04x	1.59x
	long	1423	0.63x	1.01x	1.60x

Table 6.4: LBM performance. 100 runs.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Speedup	Optimization Impact
A100	medium	1	0.78x	0.80x	1.03x
	large	18	0.58x	0.70x	1.21x
MI100	medium	13	4.19x	4.70x	1.12x
	large	28	0.65x	0.74x	1.14x

Table 6.5: OptionPricing performance. 1000 runs.

The performance of our implementation of LBM compared to the reference implementation from the Parboil benchmark suite [Str+12] can be seen in table 6.4. The datasets used are taken from Parboil. In general, short-circuiting brings the performance up to par with the hand-written implementation, even a bit higher on the MI100. The impact of the optimization is most significant on the MI100, which shows an improvement in execution time of up to 60%, while the A100 shows a more modest impact of around 10%.

6.8.5 OptionPricing

The OptionPricing benchmark is an implementation of the Black-Scholes based option pricing engine from FinPar [And+16]. The performance impact of short-circuiting can be seen in table 6.5, with datasets taken from FinPar. While short-circuiting does not quite close the gap between the two implementations, the impact of up to 21% on the A100 and up to 14% on the MI100 brings the Futhark implementation significantly closer (except for one case where the hand-written implementation is exceptionally slow on the MI100).

6.8.6 LocVolCalib

LocVolCalib (Local volatility calibration) is another implementation of a benchmark from the FinPar benchmark suite. As seen in table 6.6, short-circuiting has a modest impact here, between 4% and 12%. While the resulting performance is a bit worse than the reference implementation for the largest dataset, we show similar or even better performance for the other datasets.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Speedup	Optimization Impact
A100	small	103	0.97x	1.05x	1.08x
	medium	50	1.18x	1.27x	1.07x
	large	169	0.63x	0.68x	1.08x
MI100	small	207	1.08x	1.20x	1.12x
	medium	84	0.92x	0.97x	1.06x
	large	431	0.76x	0.79x	1.04x

Table 6.6: LocVolCalib performance. 10 runs.

	Dataset	Reference (ms)	Unoptimized Speedup	Optimized Speedup	Optimization Impact
A100	855280	70	9.82x	15.19x	1.55x
	8552800	631	76.48x	93.18x	1.22x
	85528000	6194	197.66x	208.02x	1.05x
MI100	855280	70	5.06x	6.78x	1.34x
	8552800	630	39.11x	46.08x	1.18x
	85528000	6280	115.72x	126.18x	1.09x

Table 6.7: NN performance. 100 runs.

6.8.7 NN

Finally, the NN benchmark is an implementation of K-nearest neighbors. The code consists of computing the distance between the given points using a map and a loop with a reduction inside.

With short-circuiting, Futhark is able to determine that the result of the reduce can be put directly in the result of the loop, removing the overhead of the additional copy. We see significant speedup here of up to 55% on the A100 and 34% on the MI100, as shown in table 6.7. The dataset column indicates the number of points used.

The reference implementation from Rodinia only parallelizes the initial distance computation, and not the reduction, so the Futhark implementation is significantly faster, especially as the input data gets larger.

6.9 In-place Scalar Maps

While we have so far focused on in-place updates and concatenations, there are some simple but obvious cases that we should also support. Consider for instance the following two Futhark definitions:

$$\texttt{let } f\ (xs : *[]i32) = \texttt{map}\ (+2)\ xs$$
$$\texttt{let } g\ (xs : *[]i32)\ (ys : []i32) = \texttt{map2}\ (+)\ xs\ ys$$

Notice that xs is unique in both cases, which means that a straightforward imperative implementation could reuse the memory buffer of xs for the result of the function. Short-circuiting as defined above is not able to do that, because the results of the map bodies are not arrays. However, with a bit of extra work we can handle these special cases using the ordinary short-circuiting techniques.

In short, when encountering a kernel with scalar results we can attempt to match up the return patterns with any arrays indexed inside the body of the kernel. If we can find an array that satisfies the following conditions, we can short-circuit the scalar computation and get rid of the extra allocation:

1. The array is used in an indexing operation inside the kernel.
2. The array must be last used in that indexing operation.
3. The indexing operation must only use the thread-id variables.
4. The array must have the same index function as the pattern of the kernel statement.
5. The elements of the array must have the same size as the resulting elements of the kernel.
6. The array must be unique.

If we can find such an array, we can perform short-circuiting by treating the identified array as the source and the pattern of the kernel statement as the destination. Then, regular short-circuiting can be applied to verify safety and to make the kernel call reuse the array buffer.

This tweak to short-circuiting has been implemented in the Futhark compiler, but it has negligible impact on the generated GPU code, apart from resulting more satisfying code [Hen22]. However, on the multicore backend we have observed speedups of up to 2x on simple programs like the ones above, likely due to locality.

6.10 Related Work

Many other attempts have been made at addressing the memory overhead of the functional style of languages. Destination passing style [Sha+17] adapts a region-based approach in order to attempt to create arrays directly in their destination space, but it does not use index functions and therefore does not support index-based optimization analyses like short-circuiting.

Sisal's *Build-in-Place* and *Update-in-Place* optimizations are similar to, but more limited than, short-circuiting. In particular, Update-in-Place uses dope vectors (which are similar to LMADs, see section 3.2) to model memory layout of arrays, but cannot express general layout optimizations such as coalescing.

The Data-Flow Graph Language (DFGL) [Sbî+15] uses a DSL to express dependencies between (elements of) arrays and computations in sequential code, and then uses a polyhedral framework to prove deadlock-freedom and safety of parallelization, as well as allowing certain optimizations. We take the reverse approach by starting with a program that has safe parallelism by construction, and then attempt to prove that some of the intermediate buffers (which are used conservatively to ensure safe parallelism) are not necessary. As such, we use an unrestricted language with a conventional type system to enforce separation of reads and writes. Other languages have followed the same approach as DFGL of separating program from optimizations, such as Chill [Kha+13], Halide [Rag+13] (both of which predate DFGL) and PolyMage [MVB15].

The use of LMADs for analyzing and optimizing parallelism is reminiscent of automatic parallelization of loops in Fortran [Hal+05; OR12; HPY01; RPR07; OR13]. Those analyses often aggregate loop-accesses into read-only (RO), read-write (RW) and write-first (WF) sets, expanded across loops using complex set operations like subtraction and intersection. By attacking the problem from the other direction, the operations we need to support (union and non-overlap) are much simpler to implement. Furthermore, if automatic parallelization of a given loop fails, the results are catastrophic, in that it yields sequential code. In contrast, if our analysis fails we still use all the parallelism expressed by the user, paying only a 1.1× to 2× overhead from the extra copies and allocations.

Scalarazation of array update syntax in Fortran [Zha07] is also related. They address the issue of taking an array update statement like $A(2:11) = A(1:10) + A(3:12)$ and sequentializing it in such a way that data dependencies are respected. For instance, if the above statement was naively implemented, the reads and writes from successive iterations of the resulting loop would overlap, yielding incorrect results. Scalarization is an automatic analysis and technique to correctly implement the sequential code with the fewest necessary intermediate values. However, the problem that we are trying to address is different, in that we are given an already parallel (and correct) array update statement like the one above, and trying to prove that it is safe to remove some of the overhead given by the safe parallelization.

Chapter 7

Autotuning

In this chapter, we present a technique for automatically tuning the threshold parameters that discriminate semantically-equivalent but differently-optimized code versions making up the computational kernels of a program. Assuming that the structure of the multi-versioned code adheres to a *monotonicity assumption*, our autotuner performs a one-time tuning process, relying on minimal compiler instrumentation, that generates near-optimal threshold values for all datasets represented by the tuning datasets.

7.1 Introduction

Determining the best compilation technique for a given problem can be difficult in the face of different dataset and hardware characteristics [Che+]. For instance, when writing applications with imperfectly nested parallel loops for the GPU, common wisdom tells us to use enough parallelism to saturate the hardware and sequentialize any parallelism in excess of what the hardware can support. But if the characteristics of the input datasets are allowed to vary significantly we may not be able to statically determine how much parallelism is enough. For instance, one dataset may offer a sufficient amount of parallelism in the top parallel loop to saturate the GPU, while another dataset requires exploiting one or more levels of inner parallelism as well.

As a result, even though GPUs are successfully used to power the fastest supercomputers on the planet [Sch22], they are notoriously difficult to program efficiently, especially when the problem at hand exhibits nested levels of parallelism with sizes that are statically unknown or unpredictable.

Further complicating matters, the common wisdom mentioned above does not always hold. It has been shown that even when there is enough outer parallelism to saturate the GPU, it can sometimes be more efficient to exploit additional inner levels of parallelism, for instance when that additional parallelism can be mapped to GPU workgroups and intermediate results can fit in local memory [And+16; Gie+20]. Additionally, the best optimization

strategy for one generation of GPUs may not be the best strategy for another generation of GPUs, even from the same vendor [Hen+19]. For NVIDIA GPUs, for instance, the **large** dataset from the LocVolCalib benchmark of the FinPar benchmark suite [And+16] runs faster using the common-wisdom approach on Kepler GPUs, but not on Turing GPUs where the intra-group version using workgroups and local memory is faster.

In summary, for many important applications, no common optimization recipe exists that can produce a single statically generated version of code that has optimal performance across all datasets and hardware of interest.

Much work has been done to attempt to solve this problem, such as:

1. Targeting best *average* performance across datasets by using supervised offline training methods from machine learning to infer the best configurations of compiler flags [Che+; Fur+11; Bag+15].

2. Aiming specifically at *stencil* applications by statically selecting between various compile-time optimization recipes using stochastic methods, and then using those same-shape stencils on larger arrays [Rag+13; Fra+18; Hag+18].

3. Using dynamic analysis to control the granularity for parallel multicore execution [Aca+19; TJF14].

While the last approach is infeasible on GPUs due to the runtime-system extensions needed, the first two approaches are limited by the fact that they do not attempt to classify datasets based on the best-suited code version, and are thus not able to statically construct a single program that offers optimal performance for all datasets.

One approach to addressing these concerns is to generate multi-versioned code and use runtime characteristics in combination with threshold values provided by the user to dynamically choose between versions. While this can be done in hand-written code, it is especially interesting in a functional context where syntax-based rewrite-rules can be used to automatically generate many semantically equivalent but differently optimized versions of code [Ste+].

One such technique is *incremental flattening*, a flattening transformation which is used to transform programs with many nested levels of parallelism into a form that can be efficiently executed on hardware with limited levels of parallelism, such as GPUs [Hen+19]. Incremental flattening works by mapping increasing levels of application parallelism to hardware parallelism, generating many semantically-equivalent but differently-optimized versions of code. These different versions of code, or *kernels*, are combined into one program by guarding each kernel with a predicate that compares the amount of exploited parallelism of that kernel with a *threshold value* provided by the user. In general, a program can consist of multiple such computational kernels, and each one of them can result in multi-versioned code.

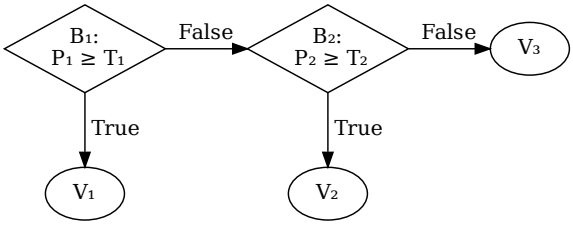


Figure 7.1: An example tuning tree.

The result is a forest of *tuning trees*[1], such as the one shown in fig. 7.1. The tuning tree in this example consists of three kernels, V_1, V_2 and V_3. At runtime, the amount of parallelism exploited by V_1, denoted P_1, is compared to the user-provided threshold value T_1. If P_1 is larger than T_1, the V_1 kernel is used for this execution. If V_1 does not exploit enough parallelism for the given dataset, P_2 (the amount of parallelism exploited by V_2) is compared to T_2. If P_2 is larger, V_2 is run, otherwise V_3 is used.

Having generated a multitude of code-versions for a particular program, the question of how to choose the best threshold values remains. Ideally, we should choose threshold values such that the best code version is always chosen for any dataset on any given piece of hardware. For large programs, manually finding the best threshold values can be infeasible due to the potentially large number of kernels generated and the opaque (for the average user) relation between the program written and the kernels generated. Additionally, even if the user knows exactly what the different generated kernels are for a given program, choosing the optimal kernel still depends on the hardware and dataset characteristics, as described above.

Instead of relying on the user to provide tuning parameters, we can employ an "autotuning" process. The idea is to define a set of tuning inputs and then use another program to systematically run our program with various combinations of tuning parameters in order to find the combination that gives the best performance for all the specified inputs, which hopefully translates to the real inputs used later on.

In the paper introducing incremental flattening [Hen+19], the authors suggest using an autotuner based on the OpenTuner framework [Ans+14] to tune the threshold parameters of incrementally flattened programs. The resulting autotuner uses a black-box approach — it has no insight into how the target program is structured — wherein the tuning parameters are initially guessed more or less randomly, relying on stochastic methods to iteratively improve each guess. Unfortunately, the search space for any given program can be huge, so this is both time consuming and does not guarantee that the found values are optimal. Indeed, our experimental evaluation in section 7.8

[1]Though the forest can consist of multiple trees, we will be considering one tree at a time, since each tree can be tuned independently.

shows that the OpenTuner-based autotuner does not always find optimal tuning parameters and that the tuning times can be unpredictable and sub-optimal for a number of real-world applications and public benchmarks.

7.1.1 Overview

In this chapter, we will describe a different approach, based on co-design between the compiler and autotuner. Inspired by incremental flattening, our technique is applicable to any kind of multi-versioned code, as long as the resulting tuning tree conforms with a *monotonicity assumption*. Defined in detail in section 7.4, the monotonicity assumption relates the relative performance of two branches of the tuning tree to the conditional discriminating between them. As an example, for the conditional B_1 in fig. 7.1 to adhere to the monotonicity assumption, it must be the case that *if* V_1 is found to be the fastest of the three kernels for some value x of P_1, *then* V_1 is also be faster than the other kernels for any larger value of P_1.

In the case of incremental flattening, the dynamic program value — P_1 in the above example — refers to the amount of parallelism exploited by the guarded kernel, e.g. V_1, but our autotuner does not require that to be the case, only that the tree conforms with the monotonicity assumption. However, we find that the trees constructed by incremental flattening adhere to the monotonicity assumption under common circumstances, and we discuss remedial techniques that could be applied in case it does not.

At the basic level, our autotuner works by finding, for each provided dataset, the set of *maximal threshold intervals* that selects the fastest kernel. If the given program adheres to the monotonicity assumption, we are then able to intersect the threshold intervals for any number of datasets into a single set of threshold intervals that selects the fastest kernel for all of those datasets and any datasets that have similar parallelism characteristics. The only requirements from the compiler is that it is possible to export the tuning tree generated by a particular program and that it is possible to show what degrees of parallelism are exploited for each kernel when run on a given dataset (e.g. what the values of P are).

The tuning tree for a given program might be traversed multiple times in the course of a single execution. For instance, an outer sequential loop around the parallel loops in the user code will cause a kernel to be selected and run multiple times. For a given dataset and program, this may cause each kernel to exploit different degrees of parallelism in each iteration (e.g. P takes on different values at different times during one execution), and it might even be the case that different kernels are preferable at different times during the same execution. We call programs where all kernels always exploit the same degree of parallelism during a single execution *size-invariant* programs and programs where some kernels exploit different degrees of parallelism during a single execution *size-variant* programs.

Our autotuning strategy is best explained in terms of size-invariant programs, but we show how size-variant programs can be considered a special-case of size-invariant programs and how to modify our autotuner to accommodate them. While size-variant programs are not fundamentally harder to tune than size-invariant programs, they can cause an explosion in the search space of threshold values, but we show how to handle that using a binary search technique.

The resulting autotuner finds the optimal threshold intervals for size-invariant programs using just $O(kd)$ runs of the program in question: Each kernel k is run once with each dataset d, in a one-time, ahead-of-time tuning process. The resulting tuning values will always pick the fastest kernel for all datasets that have similar size-characteristics as the tuning datasets. For size-variant programs, each kernel is run $O(\log m_i)$ times for each dataset, where m_i is the different degrees of parallelism exhibited by that one dataset, instead of just once.

Finally, we show the viability of our approach by applying the autotuner to Futhark programs compiled using incremental flattening, comparing against a black-box OpenTuner-based autotuner and evaluating the impact on a number of real-world applications from the remote-sensing and financial domains [Gie+20; HEO18] as well as benchmarks from public benchmark suites such as Rodinia [Che+09] and FinPar [And+16; Oan+12]. Compared with the OpenTuner-based autotuner, our method reduces the tuning time by as much as 22.6×, with an average of 6.4×. In five out of 11 cases our autotuner also finds better thresholds that speed up program runtimes by as much as 10×.

In summary, we claim the following contributions:

1. We present an autotuning technique relying on minimal compiler instrumentation that guarantees near-optimal kernel-choice when a monotonicity assumption is upheld.

2. We describe and discuss the monotonicity assumption, including proving that the monotonicity assumption guarantees that optimal interval thresholds overlap.

3. We describe how size-variant and size-invariant programs differ, and the steps required for the autotuner to be able to handle both efficiently.

4. We demonstrate the viability of our approach by applying it to Futhark programs compiled using incremental flattening and evaluating the impact on a number of real-world applications and public benchmarks.

Most of the work presented in this chapter has previously been published at TFP 2021 [Mun+21]. This thesis adds a more detailed description of the monotonicity assumption and a proof argument for the overlap of optimal threshold intervals.

7.2 Background

7.2.1 Flattening

Parallel functional array languages, such as Futhark, allow users to express arbitrary amounts of nested parallelism, for example by using the SOACs described in section 2.2: `map`, `reduce`, `scan` and `scatter`. In contrast, GPUs generally only have one or two levels of parallelism available at the hardware level. Therefore, in order to efficiently execute parallel programs on GPUs, a compiler for such a language must determine a way to "flatten" the parallelism in the given program to the level of parallelism available in the hardware. For example, a perfectly nested parallel loop `map` (`map` f) xss, where xss is a two-dimensional array, can be flattened by first flattening xss into a one-dimensional array and then applying `map` f to the resulting array. Imperfectly nested parallel loops may be flattened using flag-vectors, map-loop interchanges and other similar techniques [Ble+94].

Blelloch's transformation is a technique for "full flattening" [Ble90]. It is a universal transformation that works by fully flattening all available parallelism through a technique called vectorization, which lifts all functions f into vectorized versions $\hat{f}$ that applies to segmented arrays. Although useful because it can be generally applied, the typical shortcomings of full flattening are [Hen17]:

1. All parallelism is always exploited.

2. It does not consider communication or locality of reference. In fact, in many cases locality of reference is destroyed.

3. It may blow up memory. For instance, the fully flattened version of matrix multiplication uses $O(n^3)$ space, while the regular implementation uses $O(n^2)$.

7.2.2 Incremental Flattening

One technique that attempts to address the shortcomings of full flattening is incremental flattening [Hen+19]. It is a generic flattening algorithm that works as a top-down pass over a simple data-parallel language supporting regular nested data-parallelism, creating semantically-equivalent versions of differently-parallelized code. Each distinct code-version, or kernel, is guarded by a conditional that dynamically compares the amount of exploited parallelism of that kernel with a user-provided threshold value. If the kernel would exploit enough parallelism, that kernel is chosen.

The transformation works by recursively traversing the given code in a top-down manner, mapping parallelism to a given hardware level l. Every time a `map` is encountered it will:

```
let mapscan [m][n] (xss : [m][n]i64) : [m][n]i64 =
  map2 (\row i →
    loop row = row for _ < 64 do
      let row = map (+ i) row
      in scan (+) 0 row
  ) xss (iota m)
```

Listing 7.1: The definition of *mapscan*.

1. Produce one version of code that maps the parallelism of the discovered `map`-nest to hardware level l and sequentializes the inner parallelism of the discovered `map`.

2. Map the parallelism of the discovered map to hardware level l and recursively map the inner parallelism to hardware level $l - 1$. In the case of GPUs, this corresponds to workgroup parallelism with intermediate results stored in fast local memory.

3. Flatten the parallelism of the discovered `map` at hardware level l and recursively continue inside the body. This step essentially corresponds to applying map fission and map-loop interchange to create a perfect nest of maps that can end in a `reduce` or `scan` operation.

If we denote by V_i different kernels resulting from some function f, by $P_i(x)$ the amount of parallelism exploited by the kernel V_i for a particular dataset x and by T_i the user-defined threshold values, incremental flattening will turn a statement `let` $y = f\ x$ (where f has some amount of inner parallelism) into the following:

```
let y =
  if P_1(x) ≥ T_1 then V_1(x) else
    if P_2(x) ≥ T_2 then V_2(x) else
      ...
        if P_n(x) ≥ T_n then V_n(x) else
          V_{n+1}(x)
```

Leaving out the x, the generated code can be thought to constitute a tree of kernels, which we call a *tuning tree*, such as the one shown in fig. 7.1. By setting the threshold values appropriately for a given dataset, incremental flattening allows the user to ensure that the preferred kernel is chosen.

As an example of how incremental flattening works, consider the *mapscan* function, shown in listing 7.1.

This contrived function transforms a two-dimensional matrix by mapping over the rows and then, for each row, performing a map and a scan in a loop. In other words, *mapscan* has two levels of imperfectly nested parallelism. As a

```
let mapscan_outer [m][n] (xss : [m][n]i64) : [m][n]i64 =
  map2 (\row i →
    loop row = row for _ < 64 do
      let (row, _) =
        loop (row, acc) = (copy row, 0) for j < n do
          let tmp = acc + row[j] + i
          let row[j] = tmp
          in (row, tmp)
      in row
  ) xss (iota m)
```

Listing 7.2: The definition of $mapscan_{outer}$.

```
let mapscan_intra [m][n] (xss : [m][n]i64) : [m][n]i64 =
  -- Uses grid-level parallelism.
  map2 (\row' i →
    -- row is stored in local memory
    loop row = copy row' for _ < 64 do
      -- Uses group-level parallelism.
      let row = map (+ i) row
      in scan (+) 0 row
  ) xss (iota m)
```

Listing 7.3: The definition of $mapscan_{intra}$.

```
let mapscan_inner [m][n] (xss : [m][n]i64) : [m][n]i64 =
  loop xss = copy xss for _ < m do
    map2 (\row i →
      let row = map (+i) row
      in scan (+) 0 row
    ) xss (iota m)
```

Listing 7.4: The definition of $mapscan_{inner}$.

result, incremental flattening targeting a GPU with two levels of parallelism (taking advantage of workgroups and local memory) would produce three different versions of *mapscan*: One that exploits only the outer parallelism and sequentializes the inner `map` and `scan`, one that exploits both levels of parallelism by utilizing workgroups and keeping *row* in local memory as well as a version that fully flattens all parallelism. The resulting code, here called $mapscan_{outer}$, $mapscan_{intra}$ and $mapscan_{inner}$ are shown in listings 7.2 to 7.4

Note that the perfectly nested map and scan construction in $mapscan_{inner}$ is turned into a single flattened parallel kernel, with degree of parallelism mn. Between the first and last version, we will intuitively expect $mapscan_{outer}$ to perform well for inputs where the degree of outer parallelism, m, is large enough to saturate the GPU and $mapscan_{inner}$ to be preferable otherwise, because it exploits more parallelism. But we also have the intra-group version, which uses fast local memory for the intermediate results of the inner loop; it is likely preferable in most cases where it applies. Unfortunately, most modern GPUs, including the A100, only supports up to 1024 threads in a workgroup, so the intra-group version will only be applicable when n is less than that.

In fact, because of this extra limitation on intra-group kernels, incremental flattening will insert an extra condition on intra-group branches in the tuning tree, asserting that the amount of intra-group parallelism does not exceed that which is supported by the GPU in question. We will however elide that detail for most of this chapter, since the result is simply that the fully flattened version deepest in the tuning tree serves as a fallback for the intra-group kernels.

Thus, if we find that the intra-group version is always preferable when applicable, we can set the thresholds guarding $mapscan_{outer}$ and $mapscan_{intra}$ respectively such that it is always chosen. In terms of the tuning tree in fig. 7.1, assuming that V_1 corresponds to $mapscan_{outer}$, V_2 to $mapscan_{intra}$ and V_3 to $mapscan_{inner}$, we set $T_1 = \infty$ and $T_2 = 0$, ensuring that V_2 is always executed. This conforms with the intuitive notion that, if a given kernel is preferable for some amount of parallelism, then adding more parallelism will not hurt the relative performance of that kernel[2].

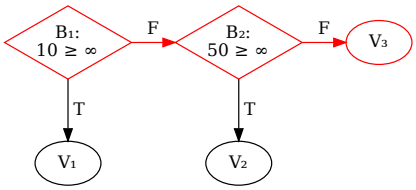
In summary, incremental flattening is a general technique that allows the compiler for a parallel array language to generate trees of code versions that are flattened differently, making it possible to choose the best version at runtime by setting a set of user-defined parameters accordingly.

Previously, finding the best threshold values for these parameters, a process called "tuning", was done using a black-box autotuner based on the OpenTuner framework [Ans+14]. Over time, and as we will see in section 7.8, this autotuner was observed to unreliable. As an alternative, we will describe our autotuner in the following sections.

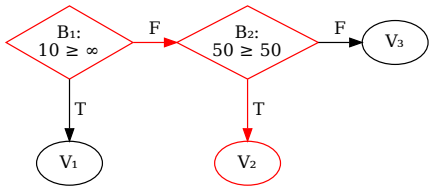
7.3 Intuition and an Example

Instead of the general black-box autotuner approach suggested by the authors of the incremental flattening paper [Hen+19], we present an autotuner based on co-design between the compiler and tuner. By adapting the tuning technique to the specific transformation producing the multi-versioned code in question, we

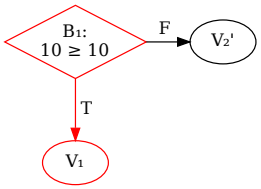
[2]We will make this notion, and its consequences, more precise when discussing the monotonicity assumption in section 7.4



(a) Running the program on V_3 by setting all thresholds to ∞.



(b) Targeting V_2 by setting $T_2 = 50$.



(c) Targeting V_1 in the collapsed tree.

Figure 7.2: Targeting specific code versions for a single dataset.

are able to take advantage of knowledge about the structure of the generated code in order to more efficiently find better threshold values.

Our framework assumes that the multi-versioned code in question has the structure of a (forest of) tree(s) such as the one in fig. 7.1. Specifically, it assumes that a number of kernels $V_1, \ldots, V_{n+1}$ are guarded by branches $B_1, \ldots, B_n$. Each branch B_i must compare a dynamic program property P_i to some user-defined threshold value T_i. If the tree conforms to a *monotonicity assumption* we guarantee that we find *near-optimal* threshold values using a small deterministic number of runs, needing only minimal compiler instrumentation.

The basic idea underlying our autotuner is to tune the threshold parameters for each dataset individually, yielding for each parameter the maximal interval which, in aggregate, will choose the fastest kernel for that dataset. Then, because of the monotonicity assumption, we are able to combine those intervals by intersecting them, yielding for each branch a single threshold interval that will choose the fastest kernel for all the tuning datasets being used. This way, we can perform a one-time tuning that will choose the fastest kernel for all datasets similar to the ones being used to tune with.

Focusing for now on the simple size-invariant case, in which the amount of parallelism exploited by each kernel does not change during a single execution, we'll use the tuning tree from fig. 7.1 as an example. Because the program is size-invariant we can reason that for a given dataset there is going to be a fastest kernel, which we should prefer. In order to find out which kernel is the fastest for a given dataset, we just need to run each kernel once on the dataset and record which is the fastest.

The steps for a single dataset is shown in fig. 7.2. We start by running the bottom-most kernel first, V_3 in this case. To run V_3, we set all thresholds to ∞

(or some sufficiently high number), as shown in fig. 7.2a. No matter what the degree of parallelism for the given dataset is at each branch, the comparison will always be false so the fall-back kernel lowest in the tree is run and we can record the execution time of that kernel.

Next, we want to target V_2 and this is where we need a bit of compiler instrumentation: If the program outputs the degrees of parallelism being compared at each branch, we can tell exactly what values to use for the threshold parameter T_2 in order to next target the V_2 kernel. In this example, P_1 was reported to be 10 and P_2 was reported to be 50. Thus, in order to run V_2 on our dataset, we set T_2 to 50 (or any value below 50), as in fig. 7.2b, and record the execution time.

Having tuned V_2 and V_3, we can consider that entire sub tree as one combined kernel formed by those two kernels. The combined kernel can be considered as one near-optimal program kernel V_2', resulting in the tree shown in fig. 7.2c, upon which we can apply recursive reasoning to run V_1.

Now that we have run all three kernels on our dataset, we can compare the execution times and determine which one is the fastest. Let's assume that V_2 is fastest. Therefore, for this dataset, the optimal threshold intervals for T_1 and T_2 are $(10, \infty)$ and $[0, 50]$. In other words, as long as T_1 is larger than 10 and T_2 is lower than or equal to 50, V_2 will be used for this particular dataset.

Assuming that the execution time of each kernel does not depend on the specific values contained in the dataset being used but only on the amount of parallelism exploited, we can use the same threshold values for all other datasets with the same size-characteristics, always running the fastest kernel. Whether it holds or not comes down to the monotonicity assumption, discussed in section 7.4.

Finally, since we have identified the individual sets of maximal optimal threshold intervals for a number of datasets we can combine the threshold intervals for each parameter by interval intersection. If such an intersection exists for each threshold, the resulting set of intervals will always select the fastest kernel for each of the datasets being used to tune with.

For example, if we found that the maximal optimal threshold intervals for two datasets x and y are:

$$T_1^x = (10, \infty) \quad \text{and} \quad T_2^x = [0, 50]$$
$$T_1^y = [0, 100] \quad \text{and} \quad T_2^y = [0, 1000]$$

Then the pairwise intersections $(10, 100]$ and $[0, 50]$ are also optimal threshold intervals for x and y (albeit not maximal).

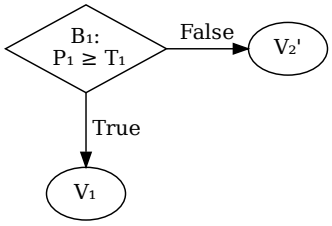


Figure 7.3: The tuning tree in fig. 7.1 with the last branch collapsed.

7.4 The Monotonicity Assumption and Correctness Argument

The autotuner we propose relies heavily on assumptions about the relative performance of code versions as a function of the amount of parallelism they exploit, as well as the general structure of the tuning tree. So far, we have relied on an intuitive understanding of this *monotonicity assumption*, given in section 7.1.1, but it is time to describe and discuss it in more detail.

Definition 7.4.1 (Monotonicity assumption)**.** For some i and some value of P_i, if the kernel V_i is found to be faster than all kernels $V_j, j > i$, then it remains faster for all larger values of P_i.

Let us consider a simple example with a tuning tree consisting of just two kernels, V_1 and V_2, with a branch $B_1 : P_1 \geq T_1$. If V_1 is measured to be faster than V_2 for a dataset with $P_1 = x$, it should also be faster for all datasets where P_1 is larger than x, no matter how V_2 is constructed. We will use this two-kernel example when discussing the monotonicity assumption further, since any larger tuning graph can be reduced to a recursive application of the two-kernel problem. For instance, assume a tuning tree with three kernels, like the one in fig. 7.1: Having tuned the threshold T_2, we can treat the entire branch of B_2 as one opaque kernel V_2', as shown in fig. 7.3, and apply recursive reasoning on the branches further up the tree.

Given such a two-kernel tuning tree, if V_1 is found to be faster than V_2 for a particular dataset d which causes V_1 to exploit an amount of parallelism x, then the maximal optimal threshold interval for that dataset and threshold parameter is $[0, x]$. Conversely, if V_2 is faster, the maximal optimal threshold interval is (x, ∞). Our tuning strategy relies on being able to take the intersection between several such maximal optimal threshold intervals, arising from different datasets. Therefore, we need to make sure that such an intersection exists.

Theorem 7.4.1 (Overlap of optimal threshold intervals)**.** *Given two datasets d_1 and d_2 and a collapsed tuning tree with kernels V_1, V_2 guarded by the branch $B_1 : P_1 \geq T_1$, if the monotonicity assumption applies then the maximal optimal threshold intervals for the two datasets overlap.*

Proof. If we denote by $R(V, d)$ the execution time of V when run on the dataset d and by $deg(V, d)$ the degree of parallelism exploited by V when run on d, there are four cases for the execution times of V_1 and V_2 when run on two datasets d_1 and d_2:

Case 1: $R(V_1, d_1) \leq R(V_2, d_1) \ \wedge \ R(V_1, d_2) \ \leq R(V_2, d_2)$
The maximal optimal threshold intervals for d_1 and d_2 are $[0, deg(V_1, d_1)]$ and $[0, deg(V_1, d_2)]$ respectively, which trivially overlap.

Case 2: $R(V_1, d_1) \leq R(V_2, d_1) \ \wedge \ R(V_1, d_2) > R(V_2, d_2)$
The maximal optimal threshold intervals for the two given datasets are $[0, deg(V_1, d_1)]$ and $(deg(V_1, d_2), \infty)$. By the monotonicity assumption it must be the case that $deg(V_1, d_1) > deg(V_1, d_2)$, so the two intervals overlap.

Case 3: $R(V_1, d_1) > R(V_2, d_1) \ \wedge \ R(V_1, d_2) \leq R(V_2, d_2)$
Symmetrical with case 2.

Case 4: $R(V_1, d_1) > R(V_2, d_1) \ \wedge \ R(V_1, d_2) > R(V_2, d_2)$
The maximal optimal threshold intervals for the given datasets are $(deg(V_1, d_1), \infty)$ and $(deg(V_2, d_2), \infty)$, which trivially overlap.

Thus, in all four cases, if the monotonicity assumption holds, the optimal threshold intervals for the two datasets overlap. □

We have thus outlined a proof that if the monotonicity assumption holds for a given branch then the optimal threshold intervals for any two datasets d_1 and d_2 have a non-empty intersection. For a given program there may be many datasets, but it follows from Helly's theorem [DGK63] that if the maximal optimal intervals for all pair-wise datasets have non-empty intersections, then the maximal optimal intervals for all datasets also have a non-empty intersection. This can also intuitively be seen by defining l_{max} as the maximal lower-bound of all the datasets and u_{min} as the minimal upper bound of all the datasets. Because the intervals of all pair-wise datasets overlap, it must also hold that $l_{max} \leq u_{min}$ and that all other intervals overlap with the interval $[l_{max}, u_{min}]$.

7.5 Limitations

Apart from the monotonicity assumption, there are several limitations to our technique. First and foremost, real-life hardware is subject to many fluctuations and variations in their performance. We have observed significant (2× and more) fluctuations in runtime across individual executions, both becoming faster due to warm-up effects and becoming slower due to throttling. Care must therefore be taken to ensure that enough runs are used to get a representative execution time in order not to get misleading tuning results. Our autotuner

implementation contains a check to ensure that the given program does not become slower after tuning each threshold, in which case the user is warned.

Furthermore, the datasets used for tuning might not be fully representative for the datasets being used later on. For instance, assume we have tuned a three-kernel program on two datasets, causing V_1 to exploit degrees of parallelism of 10 and 100, and found an optimal threshold interval of $[11; 100]$. If the monotonicity assumption holds, any dataset that causes V_1 to exploit parallelism outside of this interval will still be optimally discriminated, but we make no such promises for datasets falling inside the interval. For instance, if we find said interval of $[11; 100]$ and then run our program on a dataset with degree of parallelism 50 in V_1, we do not know if we should use V_1 or V_2. Therefore, users should strive to provide tuning datasets that are representative of the datasets to be expected in real runs, either many datasets or datasets that capture the sweet spots.

As mentioned, the monotonicity assumption does not always apply, e.g. for otherwise important parallelization concerns like tile sizes. Other approaches should therefore be used to determine such parameters. In reality, many valid tuning trees will not adhere to the monotonicity assumption, so we (or the compiler) have to be careful when constructing tuning trees. We will discuss this matter further in section 7.7.

As a result, we say that our autotuner guarantees *near-optimal* threshold values for a tuning tree in which the monotonicity assumption holds.

7.6 Tuning in Detail

Having described the intuition behind our autotuner, we now move on to a more detailed description, including the overall autotuner algorithm and a discussion about the differences between size-invariant and size-variant programs and their impact on tuning. Before continuing it is important to note that the autotuner is completely separate from the compiler: It is not able to change the structure of the tuned program or modify the code inside. It can only change the threshold values of the program and run the program with a given dataset.

7.6.1 Instrumentation

Minimal compiler instrumentation is needed to

1. Statically show the dependencies between tuning parameters of a given program.

2. Dynamically log the degree of exploitable parallelism being compared in each branch during the execution of the program.

Procedure 7.1: TUNEPROGRAM(p, $\overline{t}$, $\overline{d}$)

input : The program to run p, the thresholds in depth-first order of the tuning graph $\overline{t}$ and the training datasets $\overline{d}$.

output : Optimal threshold values for the threshold parameters in p.

```
1  t_1, ... t_n ⟵ t̄;
2  r_{t_i} ⟵ [0, ∞]   ∀i.1 ≤ i ≤ n;
3  foreach d in d̄ do
4  |  t_i ⟵ ∞, ∀i.1 ≤ i ≤ n;
5  |  bestRun ⟵ run p on d with values (t_1, ..., t_n);
6  |  foreach i in n ... 1 do
7  |  |  ([lb_i, ub_i], bestRun) ⟵ TUNETHRESHOLD(p, d, t̄, i, bestRun);
8  |  |  t_i ⟵ (lb_i + ub_i)/2;
9  |  |  r_{t_i} ⟵ r_{t_i} ∩ [lb_i, ub_i];
10 return (r_{t_1}, ..., r_{t_n});
```

7.6.2 TUNEPROGRAM

We start with TUNEPROGRAM as shown in proc. 7.1. This is the main entry-point of the autotuner, which takes as arguments a program to run, p, the n thresholds parameters of the corresponding tuning tree, $\overline{t}$, and a number of datasets $\overline{d}$. It loops over the datasets in order, and for each dataset it refines the currently found optimal threshold interval.

TUNEPROGRAM starts by initializing all threshold ranges to $[0, \infty]$ on line 2 (r_{t_i} is the optimal threshold interval found so far for T_i). Then, for each dataset, we first initialize all threshold values to ∞ (line 4) and compute the baseline execution time (*bestRun*) using the bottom-most kernel (line 5). Then, for each threshold we find the optimal threshold-interval (defined by lb_i and ub_i) along with the new best execution time using TUNETHRESHOLD (line 7), set the threshold value in question accordingly and update the global threshold range (lines 8 to 9). At the end, we return the optimal threshold intervals found (line 10).

In essence, for each dataset, our algorithm is recursively tuning the thresholds of the tuning tree like in fig. 7.2. In each iteration, the kernels further down the tree are considered as a collapsed tuning tree with some known best execution time. The kernel guarded by the current threshold is run, the execution time compared and the optimal threshold interval updated accordingly.

The TUNETHRESHOLD procedure is used to tune an individual threshold on a given dataset. In fact, there are two different versions of this procedure: One for tuning size-invariant thresholds and one for tuning size-variant thresholds. In our actual implementation of the autotuner, we always use the version aimed at variant thresholds, since there is no overhead when the threshold is in fact

invariant, but the procedure for invariant thresholds is shorter and simpler, so we will describe it here for illustrative purposes.

7.6.3 TuneInvarThreshold

Procedure 7.2: TuneInvarThreshold(p, d, $\bar{t}$, i, *bestRun*)

input : The program to run p, the training dataset d, the thresholds in the order they appear in tuning graph $\bar{t}$, the index of the current threshold to tune i and the best execution time so far *bestRun*

output : The optimal threshold interval for the dataset and threshold parameter, as well as an updated *bestRun*.

```
1  t_1, ... t_n ⟵ t̄;
2  ePar ⟵ exploitedPar(p, d, i);
3  t_i ⟵ ePar;
4  newRun ⟵ run p on d with threshold values t_1, ..., t_i, ..., t_n;
5  if newRun < bestRun then
6  |   bestRun ⟵ newRun;
7  |   lb_i ⟵ 0, ub_i ⟵ ePar;
8  else
9  |   lb_i ⟵ ePar + 1, ub_i ⟵ ∞;
10 return ([lb_i, ub_i], bestRun);
```

The pseudo code for TuneInvarThreshold is shown in proc. 7.2. The algorithm takes as inputs a program p, a dataset d, a set of threshold parameters $\bar{t}$, the index of the threshold to tune i, and the best execution time found so far *bestRun*. It starts by getting the amount of parallelism exploited by the kernel V_i on the given dataset (line 2). In the actual implementation, these parallelism values are recorded during the computation of *bestRun* on line 5 of TuneProgram by having the program output the exploited parallelism at each branch, so this is a simple lookup. Then, t_i is set to *ePar* (line 3) and is used to run the program on the given dataset (line 4). Then, depending on whether the new run was faster or slower than the previous fastest run[3], we update *bestRun* and set the upper and lower bound of the threshold and return them along with the new fastest execution time (line 10).

In other words, this simple procedure finds the threshold interval optimally discriminating a particular branch in the tuning tree for a given tuning dataset. If the monotonicity holds, then all the threshold intervals found for this particular threshold parameter will intersect and that intersection will optimally

[3]We can insert some ϵ here to adjust for fluctuations in execution time. Depending on what value is chosen for ϵ we can favor kernels further up or down the tree.

```
let mapscan_var [k] (ns : []i64)(xs : [k]i64) : [k]i64 =
  loop xs = xs for n in ns do
    let m = k/n
    let xss' : [n][m]i64 = unflatten n m xs
    let xss =
      map2 (\row i →
        loop row = row for _ < 64 do
          let row' = map (+i) row
          in scan (+) 0 row'
      ) xss' (iota n)
    in flatten_to k xss
```

Listing 7.5: The definition of $mapscan_{var}$.

discriminate the branch for all the given datasets. However, this will only handle programs that are size-invariant, meaning there is only one value of $ePar$ for given threshold and dataset. If the program or branch is size-variant, we will need additional handling.

7.6.4 TuneVarThreshold

The technique described in section 7.6.3 finds near-optimal threshold intervals for a given threshold parameter and dataset — but only if the degree of parallelism exhibited for that particular dataset is constant during a single execution. We'll now discuss the variant of TuneThreshold aimed at size-variant branches. First, we'll start with an example of what a size-variant program looks like.

In contrast to the size-invariant function *mapscan*, shown in listing 7.1, the $mapscan_{var}$ function shown in listing 7.5 is size-variant. Each iteration of the outer loop computes a 2-dimensional array of the same total size k, viewed as 2-dimensional arrays of different shapes ($n \times k/n$), which is then mapped over.

The overall structure of the resulting tuning tree, seen in fig. 7.4, is identical to the one for *mapscan* (seen in fig. 7.1), with the addition of an outer loop. This means that each threshold parameter can be compared to multiple different degrees of parallelism during a single execution of the program on a given dataset. For instance, if $ns = [1, 8, 64]$ and xs is an array of length 64, the outer loop will be run three times, each time with different values for P_1 and P_2. In the first iteration $P_1 = 1, P_2 = 64$, in the second $P_1 = 8, P_2 = 8$ and in the final iteration $P_1 = 64, P_2 = 1$. When we were talking about size-invariant programs, such as *mapscan*, we assumed that for a given dataset one kernel would always be the fastest, but that is not the case in $mapscan_{var}$: Perhaps V_2 is fastest when $P_2 = 64$ and $P_2 = 8$ but not when $P_2 = 1$. We therefore call $mapscan_{var}$ a *size-variant* program.

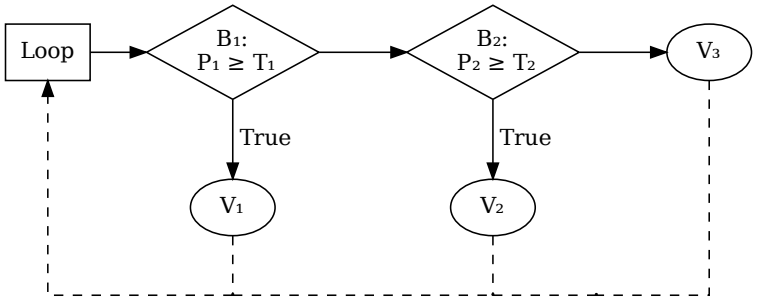


Figure 7.4: A size-variant tuning tree with an outer loop.

Fortunately, to tune $mapscan_{var}$, we can still rely on the monotonicity assumption because it is only concerned with the relationship between the structure of the tuning tree and the specific amounts of parallelism used to choose between code versions. For a particular dataset and threshold parameter, it is still the case that there is one optimal threshold value for that parameter. Continuing on the example from above, if we find that V_2 is faster than V_3 when P_2 is 8 and 64 but not when P_2 is 1, then the optimal interval for T_2 is $[8, \infty]$, for that dataset.

In fact, we can think of each instance of execution of a computation kernel during a single execution of the whole program as *separate datasets*. Even though P_2 takes on three different values during the execution of the entire loop, every time a particular kernel is executed, we are executing that kernel with a particular *instantiation* of the parallelism values. It is as if we had a size-invariant program (without the outer loop present in $mapscan_{var}$) that is run repeatedly with different inputs. Conceptually, and in terms of the monotonicity assumption, we can therefore think of each of these instantiations as separate executions. If the program respects the monotonicity assumption, there must be an optimal threshold interval for each instantiation and threshold parameter. Combining them is a simple matter of intersecting the resulting threshold intervals, just like we did for the size-invariant programs.

Likewise, when tuning two (or more) datasets, we can assume that the optimal threshold intervals for those datasets always has an intersection, because of the monotonicity assumption. This means that we don't have to change the overall tuning strategy, we just have to change how a single threshold is tuned for a given dataset.

After running the program in question on a given dataset once (with all threshold parameters set to ∞), we would be able to collect a list of all parallelism values being compared against each threshold. Thus, in principle, we could just tune each threshold against all values as if they were separate datasets. For instance, in the example above, we would first set $T_2 = \infty$, to see that P_2 takes on the values $1, 8, 64$. To find the best threshold interval for the parallelism value 64 we would set $T_2 = 64$. If the resulting run is faster

than the previous run (because one of the iterations of the main loop used V_2 instead of V_3 and V_2 turned out to be faster when $P_2 = 64$, whereby the whole execution would be faster), we would know that the optimal threshold interval for that instantiation is $[0, 64]$. Then we could continue tuning the next instantiation by setting $T_2 = 8$ yielding a new interval that could be intersected with the previous one, and so on.

The problem with size-variant branches and programs is that they can create an explosion of different parallelism values to tune against. For instance, one could easily create a program that causes the parallelism of a particular branch to span the entire space of 64-bit integers, which is of course infeasible to tune naively. This could in principle also happen with size-invariant programs, but it is unlikely because the user would have to submit as many datasets to the tuning process.

Instead, we can take advantage of the monotonicity assumption: Assume a collapsed tuning tree like in fig. 7.3, and assume that P_1 takes on n distinct values $x_1, \ldots, x_n$ (in sorted order) during the execution of a given dataset d. Assume further that we have run the program two times with $T_1 = x_j$ and $T_1 = x_{j+1}$ respectively, and that the first run was faster. The only difference between the two runs is that the first run uses the kernel V_1 when $P_1 = x_j$ while the second run uses V_2. All other kernel uses are the same between the two runs.

Because the first run is faster, V_1 must be faster than V_2 when $P_1 = x_j$. Therefore, by the monotonicity assumption, it follows that V_1 will remain faster for any value higher than x_j, so the optimal threshold value for this dataset must lie in the interval between x_j and x_n. Conversely, if the second run was faster, the optimal value would be somewhere between x_1 and x_j.

Therefore, if the monotonicity assumption holds, we can use a binary search to find the optimal value (or interval) of T_i. By performing a binary search of the n possible threshold values for a given dataset and threshold parameter we can find the optimal interval in $O(\log n)$ runs instead of $O(n)$ runs.

The resulting variation of TuneThreshold is called TuneVarThreshold and can be seen in proc. 7.3. It takes the same arguments as TuneInvarThreshold, but it uses a binary search to find the best threshold interval for the given dataset and threshold parameter. It uses a simple helper procedure called minInd which is shown in proc. 7.4.

TunevarThreshold works as follows: First, the list of exploited parallelism values is computed (line 2). The list is supposed to be sorted and deduplicated[4]. On the next line, that list is augmented with 0 and ∞ in either end, making sure that we represent all possible splits between the two guarded kernels. Next we compute the execution times of the outer boundaries of the tuning space: r_{low} represents the execution time found at the lowest end of

[4]Like in TuneInvarThreshold, these values are actually precomputed, so this is a simple lookup.

Procedure 7.3: TUNEVARTHRESHOLD($p, d, \bar{t}, i, bestRun$)

input : The program to run, p; the current dataset, d; the thresholds in the order they appear in the tuning graph, $\bar{t}$; the index of the current threshold to tune, i; and the best execution time so far, $bestRun$.

output : Optimal threshold values for the threshold parameters in p.

```
t_1, ..., t_n ⟵ t̄;
ePar_1, ..., ePar_m ⟵ SortedExploitedPar(p, d, t_i);
ePar̄ ⟵ 0, ePar_1, ..., ePar_m, ∞;
low ⟵ 0;
r_low ⟵ execution time of p on d with t_i set to 0;
high ⟵ m + 1;
r_high ⟵ bestRun;
(bestRun, bestInd) ⟵ minInd(r_low, low, r_high, high);
while low < high do
    mid ⟵ ⌊(low + high)/2⌋;
    r_mid ⟵ execution time of p on d with t_i set to ePar_mid;
    if r_high < r_mid then
        low ⟵ mid + 1
    else
        if r_low < r_mid then
            high ⟵ mid − 1
        else
            r_grd ⟵ execution time of p on d with t_i set to ePar_{mid+1};
            if r_mid < r_grd then
                (bestRun, bestInd) ⟵
                  MinInd(bestRun, bestInd, r_mid, mid);
                high ⟵ mid − 1;
            else
                (bestRun, bestInd) ⟵
                  MinInd(bestRun, bestInd, r_grd, mid + 1);
                low ⟵ mid + 2;
(lb_i, ub_i) ⟵ (ePar_{bestInd−1} + 1, ePar_{bestInd+1} − 1);
return ([lb_i, ub_i], bestRun);
```

Procedure 7.4: MinInd($r_{low}, low, r_{high}, high$)

input : Execution times r_{low} and r_{high} and their corresponding indices, low and $high$.
output : The fastest execution time and the corresponding index.

```
1 if r_low < r_high then
2 |  return (r_low, low);
3 else
4 |  return (r_high, high);
```

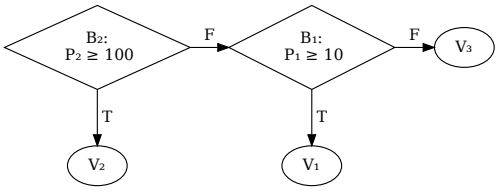
the tuning space and is computed on line 5, r_{high} is the execution time at the highest end of the tuning space and is initialized to the value of *bestRun*. The initial *bestRun* is assumed to contain the execution time when $t_i = \infty$, so we don't need to recompute it again (line 8). These will serve as the outer limits of our binary search interval, which is then iteratively refined in the main loop. After the loop, *bestRun* and *bestInd* are updated to contain the best execution time found so far and the corresponding index.

Inside the loop, we first find the index of the halfway point in the current tuning space and run the program using that value as the threshold value, recording the resulting execution time (lines 10 to 11), after which the lower and upper bound is refined. In the two simple cases, when either of the higher or lower execution times are faster than the newly found execution time, we simply update *high* or *low*, narrowing the search space, and loop. Otherwise, if the middle run was faster than the runs at the current boundaries, we run the program again using a neighboring point to get a gradient of the execution time relative to the value of the threshold parameter (line 18). This gradient is then used to refine the search space (lines 19 to 24). All in all, for a given dataset and threshold, TunVarThreshold will find the optimal threshold value using $O(\log n)$ runs of the program, where n is the number of different amounts of parallelism exhibited used at that threshold.

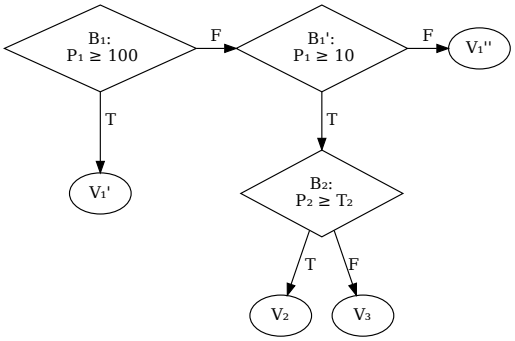
7.7 Broken Monotonicity

The monotonicity assumption does not hold for all possible tuning trees. In fact, it is easy to construct a tuning tree that is not monotonic, for instance by changing the comparison operator or swapping kernels around in our *mapscan* example. However, if our compiler carefully constructs the tuning tree such that the monotonicity assumption is adhered to, we will be able to use our autotuning framework to tune the threshold parameters.

Our experimental evaluation in section 7.8 indicates that incremental flattening in many common cases produces tuning trees that are amenable to our autotuning techniques, but that is not guaranteed. For instance, if one were



(a) Swapping B_1 and B_2.



(b) Adding duplicate code versions.

Figure 7.5: Alternative versions of the tuning graph, enabling different constraints.

to construct a program with a tuning tree like the one in fig. 7.1 that had the property that V_1 is fastest whenever the amount of inner parallelism is less than some constant k and V_2 is faster otherwise, it would not adhere to the monotonicity assumption.

In other words, although incremental flattening often produces tuning trees that can be tuned using our autotuner framework, the expressive power of the resulting trees are limited to relatively simple conditions.

We can handle some of these cases by modifying the tuning tree. For instance, we might be able to change the order of kernels or duplicate kernels, as in fig. 7.5, to construct a tree that is monotonic. Incremental flattening does not do this automatically, and it is an open question whether it could be done.

7.8 Experimental Results

For experimental validation of our autotuning framework, we have implemented it for the Futhark programming language, which uses incremental flattening to construct its tuning trees. We compare the tuning time and performance impact of using our autotuner against using the old OpenTuner-based black-box tuner proposed in [Hen+19].

7.8.1 Hardware and Methodology

All tuning and benchmarking reported here is performed using a NVIDIA GeForce RTX 2080 Ti GPU, but we have observed similar results on a slightly older GTX 780 Ti.

By virtue of being the only tuning solution available in Futhark for a while, the OpenTuner implementation has been highly optimized. It will, among other things, use memoization techniques to avoid running the same combination of kernels twice, attempting to minimize the number of runs. However, it is still inherently random and measuring GPU performance is itself subject to execution time fluctuations, so the tuning time and resulting threshold values can vary significantly.

In order to get representative results, we base our analysis of each benchmark on three separate passes. In each pass we:

1. Benchmark the untuned program 500 times and record the best execution time found.

2. Use the OpenTuner-based autotuner to tune the threshold values, run the benchmark 500 times using the thresholds found and record the best execution time found.

3. Use our autotuner to tune the threshold values, run the benchmark 500 times using the thresholds found and record the best execution time found.

In all cases, we use the same datasets for tuning and for the actual benchmarking. We report the best of the three tuning times for both OpenTuner and the autotuner. As for execution times: we report the best of three execution times found for untuned, both the worst and best for the OpenTuner-based autotuner and only the worst execution time for our autotuner.

7.8.2 Benchmarks and Datasets

We now describe the benchmarks used to show the impact of our autotuner. The datasets and number of thresholds for each benchmark are shown in table 7.1. The D1 and D2 columns show the two different datasets used and the Thresholds column shows the number of threshold parameters to tune. The implementations for all benchmarks are taken from the `futhark-benchmarks` repository [Hac].

Heston is a calibration model for the Hybrid Stochastic Local Volatility / Hull-White model [HEO18]. We use datasets from the `futhark-benchmarks` repository [Hac].

BFAST [Gie+20] is used to measure changes in satellite time-series data in order to detect e.g. deforestation, and is widely used in the remote sensing

Benchmark	D1	D2	Thresholds
Heston	1062 quotes	10000 quotes	9
BFAST	peru	africa	16
LocVolCalib	medium	large	2
OptionPricing	small	large	1
LUD	256×256	2048×2048	9
Backprop	2^{14}	2^{20}	1
LavaMD	$10^3 \times 50$	$3^3 \times 50$	4
NW	2048×2048	1024×1024	6
NN	1×855280	4096×128	3
SRAD	$1 \times 502 \times 458$	$1024 \times 16 \times 16$	4
Pathfinder	$1 \times 100 \times 10^5$	$391 \times 100 \times 256$	1

Table 7.1: Benchmarks and the datasets used to experimentally validate our autotuner.

community. For our benchmarks, we use datasets from the `futhark-kdd19` repository [OH].

LocVolCalib (local volatility calibration) and OptionPricing are implementations of real-world financial computations from the FinPar benchmark suite [And+16; Oan+12]. Here, we use datasets from the `finpar` repository [Oan+].

LUD, Backprop, LavaMD and NN are all benchmarks from the Rodinia benchmark suite [Che+09]. For these benchmarks, we use datasets from Rodinia except in cases (such as Backprop) where Rodinia only has one dataset, in which case we have created datasets that exhibit different parallelization characteristics from the original dataset. The inputs are matrices or arrays of the sizes indicated in D1 and D2. The LUD benchmark is the only benchmark that is size-variant.

The NW, SRAD and Pathfinder benchmarks are also from Rodinia, but we have implemented batched versions in order to have more nested levels of parallelism: The outer dimension in the dataset description indicates the number of matrices used as input.

Table 7.2 shows the tuning time and tuning time speedup for each benchmark. In general, we see a significant reduction in tuning time, from 1.4× for LUD to 22.6× for Heston. Without those two outliers, the average speedup is 5.1×.

More thresholds generally leads to higher tuning times, but other factors also have an effect, such as execution time of individual runs, and whether a benchmark is size-variant or not. The LUD benchmark shows the least improvement in tuning time because it has size-variant parallelism, causing the autotuner to have to use a binary search to find the optimal threshold values. Therefore, seemingly, our implementation is only a bit faster than the

Benchmark	Opentuner (s)	Autotuner (s)	Tuning Speedup
Heston	3798	168	22.59x
BFAST	1127	206	5.47x
LocVolCalib	101	21	4.83x
OptionPricing	31	6	5.40x
LUD	611	430	1.42x
Backprop	30	8	3.65x
LavaMD	104	28	3.67x
NW	222	29	7.62x
NN	125	36	3.48x
SRAD	148	28	5.31x
Pathfinder	66	10	6.81x

Table 7.2: Tuning-time and speedup between the OpenTuner implementation and our autotuner.

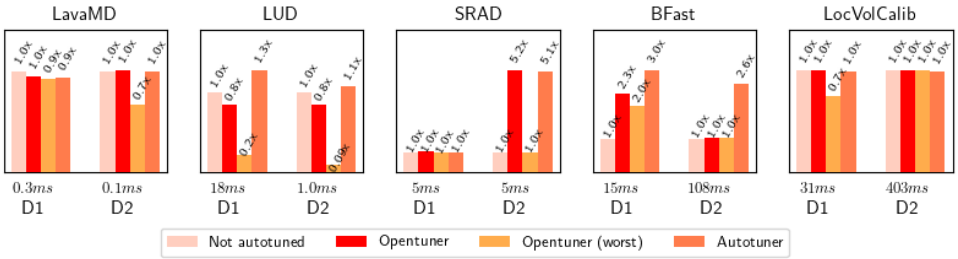


Figure 7.6: Benchmark execution time speedup. The baseline is untuned performance. Higher is better.

OpenTuner-based tuner. But as we will see later, due to the random nature of the OpenTuner-based autotuner, it sometimes finds degenerate threshold values leading to very bad performance of the tuned program.

Not shown here is the fact that our autotuner has very low variance in tuning time compared to the OpenTuner-based one, because our autotuner is deterministic: It will always perform the same number of runs necessary to tune a particular dataset. For instance, it took the OpenTuner-based autotuner between 366 and 881 seconds to tune LUD, while our autotuner used roughly the same amount of time each time. Thus, using our autotuner, the programmer is able to reason about tuning time, and can weigh whether to add additional training datasets or not.

Figure 7.6 shows the execution time speedup of programs tuned using OpenTuner and our autotuner compared to untuned execution. We have chosen to focus on the LavaMD, LUD, SRAD, BFAST and LocVolCalib benchmarks, since they are the only ones whose performance characteristics change significantly when tuned. The Futhark compiler uses sensible default thresholds and various heuristics (such as favoring intra-group parallelism when possible) to ensure

that even untuned programs have good performance. For all benchmarks, our autotuner finds the best threshold values (the slight variation in D1 of LavaMD can be chalked down to performance fluctuations), conforming with our claim that they are in fact near-optimal.

LUD, having size-variant parallelism and being very sensitive to the choice of threshold values shows some of the worst performance for OpenTuner. We see that while the untuned version performs reasonably well and the autotuner consistently finds the optimal threshold values, OpenTuner is struggling. Because of the inherently random process, the best thresholds it finds only lead to 80% the performance of the untuned version, while in the worst case performance grinds to a halt.

OpenTuner sometimes finds the correct threshold values in LavaMD, SRAD and LocVolCalib, but we also see that it is inconsistent. In the worst case it might settle on threshold values that are significantly worse than the optimal, as in the second dataset of SRAD, or even default values, as in the second dataset of LavaMD.

Like LUD, BFAST relies on intra-group parallelism and is highly sensitive to the choice of threshold values. It therefore receives a significant boost in performance from accurate tuning, such as the one produced by our autotuner. In contrast, the OpenTuner tool cannot even handle the largest dataset (africa), because some code versions cause the GPU to run out of local memory, for which the OpenTuner-based autotuner has no fallback strategy. Our autotuner correctly identifies which code version causes the problem and is able to avoid it for the rest of the tuning. The result is that, while OpenTuner performs reasonably well on the smaller dataset (though not as well as our autotuner), it fails to tune the largest dataset at all, leaving lots of performance on the table.

7.9 Related Work

A preliminary demonstration of a deterministic autotuner based on monotonicity properties was investigated in a MSc thesis by Svend Lund Breddam [Bre19] and shown to compare favorably to other AI-based approaches.

Other related work in autotuning is organized in one of three groups:

The first group of autotuners work by inferring compilation flags based on *best average performance* across training datasets for some given hardware. Typically, this is implemented using some kind of machine learning, i.e. relying on supervised offline training [Fur+11]. These methods have yielded promising result on multi-core [Che+] and many-core [Bag+15] systems, for instance by inferring compiler flags for GCC or finding near-optimal tile sizes for GPU code generation.

Secondly, languages such as Lift [Ste+; Hag+18] and SPIRAL [Fra+18] have used the method of generating different versions of code using the rich rewrite-rule systems of functional languages, and then having a tuning process

to choose between them on a per-dataset and per-hardware configuration basis. Using this method, they are able to maximize performance for that particular combination of dataset and hardware, but will have to retune if the dataset changes. Similarly, Halide [Rag+13] finds the best way to fuse image-processing pipelines using stochastic methods, resulting in various combinations of tiling, sliding windows and work replication transformations. Again, tuning is done on a per-dataset basis, but inferred tile sizes may be portable to larger datasets.

Finally, black-box approaches such as OpenTuner comprises a third group of autotuners. They use stochastic search strategies like hill-climbing and simulated annealing, as well as allowing the user to specify custom search procedures. ATF [RG19] uses a similar approach by allowing the user to write annotations in their language to help guide the autotuning framework. These approaches can work well when the entire tuning-space provides new information to the tuner, but are hindered by the inherent lack of insight into how the program actually is constructed. When the search space is vast, and only a limited amount of points actually provide new information for the tuner, as is the case in incrementally flattened programs, these approaches can take a long time to find the right tuning parameters, if they ever do.

Our autotuner combines some of these approaches by taking a compiler and autotuner co-design approach. By having the compiler output a minimal amount of data about the generated program structure and by relying on assumptions about how that structure relates to the performance of the individual generated kernels we are able to guarantee near-optimal performance for all datasets, given representative training datasets. This contrasts to the first approach, which only targets average performance, and the second approach, where you have to retune for every dataset. We are also able to perform that tuning in a small, predictable amount of time, which differentiates us from the third group.

Chapter 8

Conclusions and Future Work

We have presented static and dynamic analyses aimed at improving the performance of GPU-oriented code. In particular, we have:

1. Described how LMADs can be used to represent array slicing in programming languages, allowing more expressive slices such as staggered or diagonal blocked views of an array.

2. Described how LMADs can be used as index functions in the intermediate representation of compilers, allowing the compiler to statically transform change-of-layout operations into zero-overhead operations at runtime.

3. Presented a non-overlapping test for multi-dimensional LMADs (in the set interpretation) that allows self-overlapping dimensions within the same LMAD, along with a demonstration of how to apply it to the complex NW benchmark.

4. Formally described a series of intermediate representations, Fun, FunMem and Imp, aimed at allowing a compiler for a parallel array language to non-semantically introduce memory without losing the high-level context.

5. Shown a series of analyses and optimizations based on the FunMem imperative language, chief among them the array short-circuiting algorithm, which allows the compiler to automatically optimize in-place updates and concatenations to reduce the amount of copying and memory overhead arising from e.g. from high-level language guarantees. All these optimizations have been implemented in the Futhark compiler, with short-circuiting showing significant speedups in the generated code, leading to performance that is competitive with hand-written code.

6. Presented a technique for automatically finding near-optimal threshold values for the parameters of a tuning tree for multi-versioned code that adheres to the monotonicity assumption. We have also implemented this

technique for the Futhark programming language and experimentally validated it against a black-box tuning approach, showing significant improvements in both tuning time and performance of the tuned program.

While the last point is slightly separate from the rest (it is dynamic, the rest are static), they share the context: Automatically generating code that efficiently executes on the GPU.

8.1 Limitations and Future Work

While LMAD-slices have proven valuable in implementing in-place versions of both the NW and LUD benchmarks, we have not found any use for their rich expressiveness elsewhere. It would be interesting to investigate whether there are any other applications with complex indexing and slicing operations that could benefit from LMAD-slices. Furthermore, LMAD-slices are currently only implemented as an intrinsic in Futhark that is unsuitable for use by the average programmer. More work needs to be done to expose them in a safe manner, in order to ensure that e.g. self-overlapping LMAD-slices cannot be written to, and to provide a friendly user-experience.

Array short-circuiting, while powerful, also harbors potential for improvements. For example, we currently limit short-circuit attempts to cases where the source is last used at the short-circuit point. It might be possible to loosen this restriction in some cases. For instance, consider the following code:

```
...
let ys[k] = x -- Short-circuit point, x is source
let z : int = reduce (+) 0 x
```

Although x is used after the short-circuit point, we are only reading from it and ys has not been further altered, so it should still be safe to perform short-circuiting. Unfortunately, our analysis does not presently handle such cases.

Similarly, we only support reversing a limited set of change-of-layout operations. As an example, we do not currently support short-circuiting x' into ys in the following code:

```
let x : [n]@x_mem → L = ...
let x' : [m]@x_mem → slice(L, [0:2m:2]) = x[0:2m:2]
let ys[k] = x
```

However, if we can prove that only the elements of x that are included in x' are ever used, it should still be safe to short-circuit x' into ys. The result would be that the unused elements of x would not be computed, and so x would be able to fit directly in the space of ys.

Many of the papers on automatic parallelization of sequential loops [RHR03; Rus+06] use augmented runtime versions of LMADs called RT_LMADs or USR. RT_LMADs allow for much more precise access summaries, by including operations like subtraction and intersection, at the cost of increased complexity. They also shift the analysis more in the direction of dynamic analysis, which is not as suitable in the context of GPU execution, but it might be worth exploring whether there are parts that make sense for array short-circuiting.

The primary limitation of our autotuning work is the monotonicity requirement. Although we have demonstrated that our technique applies to a host of benchmark programs, the monotonicity assumption is not guaranteed to hold for all such programs. One avenue of future work is to investigate whether we can automatically detect such cases and take steps to improve the situation by modifying the tuning tree, as described in section 7.7. This would allow our autotuner to apply to even more programs.

Bibliography

[Aca+19] Umut A. Acar et al. "Provably and Practically Efficient Granularity Control". In: PPoPP '19. 2019, pp. 214–228. DOI: 10.1145/3293883.3295725.

[And+16] Christian Andreetta et al. "Finpar: A parallel financial benchmark". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.2 (2016), pp. 1–27.

[Ans+14] Jason Ansel et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: *International Conference on Parallel Architectures and Compilation Techniques*. 2014. URL: http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf.

[Bag+15] Riyadh Baghdadi et al. "PENCIL: a platform-neutral compute intermediate language for accelerator programming". In: *2015 PACT*. 2015, pp. 138–149.

[BG95] Guy Blelloch and John Greiner. "Parallelism in sequential functional languages". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. 1995, pp. 226–237.

[Ble+94] Guy E Blelloch et al. "Implementation of a portable nested data-parallel language". In: *Journal of parallel and distributed computing* 21.1 (1994), pp. 4–14.

[Ble90] Guy E Blelloch. *Vector models for data-parallel computing*. Vol. 2. MIT press Cambridge, 1990.

[Bon+08] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595.

[Bre19] Svend Lund Breddam. "Futhark Autotuners for Incremental Flattening". In: (2019).

[Cha+81] Gregory J Chaitin et al. "Register allocation via coloring". In: *Computer languages* 6.1 (1981), pp. 47–57.

[Che+] Yang Chen et al. "Evaluating Iterative Optimization Across 1000 Datasets". In: PLDI '10, pp. 448–459. DOI: 10.1145/1806596.1806647.

[Che+09] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[CSS12] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. "Expressive array constructs in an embedded GPU kernel programming language". In: *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*. 2012, pp. 21–30.

[CSS15] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. "Polyhedral Optimizations of Explicitly Parallel Programs". In: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. PACT '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 213–226. ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.44. URL: https://doi.org/10.1109/PACT.2015.44.

[DGK63] Ludwig Danzer, Branko Grünbaum, and Victor Klee. "Helly's theorem and its relatives". In: *Convexity*. Ed. by Victor Klee. Vol. 7. Proceedings of Symposia in Pure Mathematics. American Mathematical Society. 1963, pp. 101–180.

[Fra+18] Franz Franchetti et al. "SPIRAL: Extreme Performance Portability". In: *Proceedings of the IEEE* 106 (2018), pp. 1935–1968.

[Fur+11] Grigori Fursin et al. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". In: *International Journal of Parallel Programming* (2011), pp. 296–327.

[Gie+20] Fabian Gieseke et al. "Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values". In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 2020, pp. 385–396. DOI: 10.1109/ICDE48307.2020.00040.

[GW78] Leo J Guibas and Douglas K Wyatt. "Compilation and delayed evaluation in APL". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1978, pp. 1–8.

[Hac] The Futhark Hackers. *futhark-benchmarks*. URL: https://github.com/diku-dk/futhark-benchmarks.

[Hag+18] Bastian Hagedorn et al. "High Performance Stencil Code Generation with Lift". In: ACM, 2018, pp. 100–112. DOI: 10.1145/3168824.

[Hal+05] Mary W. Hall et al. "Interprocedural Parallelization Analysis in SUIF". In: *Trans. on Prog. Lang. and Sys. (TOPLAS)* 27(4) (2005), pp. 662–731.

[Hen+17] Troels Henriksen et al. "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*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[Hen+19] Troels Henriksen et al. "Incremental Flattening for Nested Data Parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 53–67. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295707. URL: http://doi.acm.org/10.1145/3293883.3295707.

[Hen17] Troels Henriksen. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[Hen22] Troels Henriksen. *In-place mapping and the pleasure of beautiful code nobody will ever see*. 2022. URL: https://futhark-lang.org/blog/2022-12-06-in-place-map.html.

[HEO18] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. "Modular Acceleration: Tricky Cases of Functional High-Performance Computing". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2018. St. Louis, MO, USA, 2018, pp. 10–21. DOI: 10.1145/3264738.3264740.

[Hij+22] Pieter Hijma et al. "Optimization Techniques for GPU Programming". In: *ACM Computing Surveys* (2022).

[Hoe98] Jay Philip Hoeflinger. *Interprocedural parallelization using memory classification analysis*. University of Illinois at Urbana-Champaign, 1998.

[HPY01] Jay Hoeflinger, Yunheung Paek, and Kwang Yi. "Unified Interprocedural Parallelism Detection". In: *International Journal of Parallel Programming* 29(2) (2001), pp. 185–215.

[Jan+10] Byunghyun Jang et al. "Exploiting memory access patterns to improve memory performance in data-parallel architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (2010), pp. 105–118.

[Kar72] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Springer. 1972, pp. 85–103.

[Kha+13] Malik Khan et al. "A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013). ISSN: 1544-3566. DOI: 10.1145/2400682.2400690. URL: https://doi.org/10.1145/2400682.2400690.

[Mac+19] Dougal Maclaurin et al. "Dex: array programming with typed indices". In: *Program Transformations for ML Workshop at NeurIPS 2019*. 2019.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[MH99] Sungdo Moon and Mary W. Hall. "Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization". In: *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*. 1999, pp. 84–95.

[MOHss] P. Munksgaard, C. Oancea, and T. Henriksen. "Compiling a functional array language with non-semantic memory information". In: *IFL 2022* (in press).

[Mun+21] Philip Munksgaard et al. "Dataset sensitive autotuning of multi-versioned code based on monotonic properties". In: *International Symposium on Trends in Functional Programming*. Springer. 2021, pp. 3–23.

[Mun+22] P. Munksgaard et al. "Memory Optimizations in an Array Language". In: *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2022, pp. 424–438. URL: https://doi.ieeecomputersociety.org/.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 429–443. ISBN: 9781450328357. DOI: 10.1145/2694344.2694364. URL: https://doi.org/10.1145/2694344.2694364.

[Nvi22] Nvidia. *A100 Tensor Core GPU architecture*. 2022.

[Oan+] Cosmin Oancea et al. *finpar*. URL: http://github.com/HIPERFIT/finpar.

[Oan+12] Cosmin E. Oancea et al. "Financial Software on GPUs: Between Haskell and Fortran". In: *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC '12. Copenhagen, Denmark, 2012, pp. 61–72. DOI: 10.1145/2364474.2364484.

[OH] Cosmin Oancea and Troels Henriksen. *futhark-kdd19*. URL: https://github.com/diku-dk/futhark-kdd19.

[OR12] Cosmin E Oancea and Lawrence Rauchwerger. "Logical inference techniques for loop parallelization". In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 2012, pp. 509–520.

[OR13] Cosmin E. Oancea and Lawrence Rauchwerger. "A Hybrid Approach to Proving Memory Reference Monotonicity". In: *Languages and Compilers for Parallel Computing*. Ed. by Sanjay Rajopadhye and Michelle Mills Strout. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 61–75. ISBN: 978-3-642-36036-7.

[OR15] Cosmin E Oancea and Lawrence Rauchwerger. "Scalable conditional induction variables (CIV) analysis". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 213–224.

[Pae97] Y Paek. "Automatic parallelization for distributed memory machines based on access region analysis". PhD thesis. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, 1997.

[PHP98] Yunheung Paek, Jay Hoeflinger, and David Padua. "Simplification of array access patterns for compiler optimizations". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming language design and implementation*. 1998, pp. 60–71.

[PS99] Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.5 (1999), pp. 895–913.

[Rag+13] Jonathan Ragan-Kelley et al. "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: 10.1145/2491956.2462176.

[RG19] Ari Rasch and Sergei Gorlatch. "ATF: A generic directive-based auto-tuning framework". In: *Concurrency and Computation: Practice and Experience* 31.5 (2019), e4423.

[RHR03] Silvius Rus, Jay Hoeflinger, and Lawrence Rauchwerger. "Hybrid Analysis: Static & Dynamic Memory Reference Analysis". In: *Int. Journal of Par. Prog* 31(3) (2003), pp. 251–283.

[RPR07] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. "Sensitivity Analysis for Automatic Parallelization on Multi-Cores". In: *Procs. Int. Conf. on Supercomp.* 2007, pp. 263–273.

[Rus+06] Silvius Rus et al. "Region Array SSA". In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. 2006, pp. 43–52.

[Sat61] Kirk Sattley. "Allocation of storage for arrays in ALGOL 60". In: *Communications of the ACM* 4.1 (1961), pp. 60–65.

[Sbî+15] Alina Sbîrlea et al. "Polyhedral Optimizations for a Data-Flow Graph Language". In: *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*. LCPC 2015. Raleigh, NC, USA: Springer-Verlag, 2015, pp. 57–72. ISBN: 9783319297774. DOI: 10.1007/978-3-319-29778-1_4. URL: https://doi.org/10.1007/978-3-319-29778-1_4.

[Sch22] David Schneider. "The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second". In: *IEEE Spectrum* 59.1 (2022), pp. 34–35.

[Sha+17] Amir Shaikhha et al. "Destination-Passing Style for Efficient Memory Management". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 12–23. ISBN: 9781450351812. DOI: 10.1145/3122948.3122949. URL: https://doi.org/10.1145/3122948.3122949.

[Ste+] Michel Steuwer et al. "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code". In: ICFP 2015, pp. 205–217. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784754.

[Str+12] John A Stratton et al. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* 127 (2012).

[TB98] Mads Tofte and Lars Birkedal. "A region inference algorithm". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.4 (1998), pp. 724–767.

[TJF14] Peter Thoman, Herbert Jordan, and Thomas Fahringer. "Compiler Multiversioning for Automatic Task Granularity Control". In: *Concurr. Comput. : Pract. Exper.* 26.14 (2014), pp. 2367–2385. DOI: 10.1002/cpe.3302.

[Yan+10] Yi Yang et al. "A GPGPU compiler for memory optimization and parallelism management". In: *ACM Sigplan Notices* 45.6 (2010), pp. 86–97.

[Zha07] Yuan Zhao. "Array syntax compilation and performance tuning". PhD thesis. Rice University, 2007.