Aalto-yliopisto
Aalto-universitetet
Aalto University

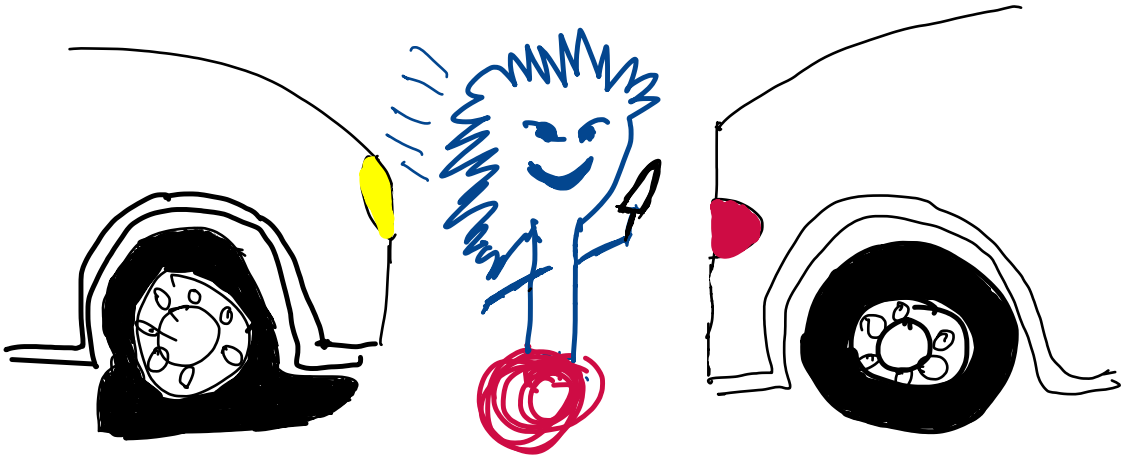
DTU

UNIVERSITY OF COPENHAGEN

Master's Programme in Security and Cloud Computing

Full Flattening of Nested Data Parallelism in the Futhark Compiler

Amirreza Hashemi



Master's Thesis
2026

Aalto-yliopisto
Aalto-universitetet
Aalto University

DTU

UNIVERSITY OF COPENHAGEN

Author Amirreza Hashemi

Title Full Flattening of Nested Data Parallelism in the Futhark Compiler

Degree programme Security and Cloud Computing

Major Security and Cloud Computing

Thesis supervisor Dr Jukka Suomela, Dr Alceste Scalas

Thesis advisor Dr Troels Henriksen

Collaborative partner University of Copenhagen

Date 16 July 2026 **Number of pages** 85 **Language** English

Abstract

Nested data parallelism is a natural way to express many algorithms, including irregular computations such as sparse-matrix and graph algorithms, but GPUs efficiently support only flat parallelism. This thesis develops a full flattening pass for the Futhark compiler that transforms irregular nested parallelism into flat GPU operations while avoiding many of the classical costs of flattening. The pass uses the uniformity of computations with respect to the surrounding map nest to generate efficient regular code whenever possible, while using the general transformation of full flattening only for genuinely irregular computations. It further avoids distributing scalar computations and leaves the replication of variant free variables implicit. The pass is also integrated with incremental flattening, allowing the compiler to choose at run time between several semantically equivalent versions of a map nest. The implementation covers the full Futhark language and passes the complete Futhark test suite, including additional tests for irregular parallelism. It also allows the GPU backend to compile programs whose intermediate shapes are discovered only at run time and that were previously rejected. On regular workloads, the pass mostly preserves the performance of the existing compiler, with a geometric mean speedup of 0.99 on the Futhark benchmark suite. On the evaluated synthetic sparse-matrix–vector multiplication workloads, the natural nested implementation is 296× faster on average than when compiled by the existing compiler.

Keywords flattening transformation, nested data parallelism, Futhark, functional array programming, GPU compilation

Co-funded by the European Union

Contents

Chapter 1

Introduction

Modern processor performance improvements increasingly rely on parallelism rather than on increasing clock frequencies. Improvements in sequential execution have slowed considerably, while processors provide ever greater parallel execution capacity, of which the Graphics Processing Unit (GPU) is the most notable mainstream example. A modern GPU can execute tens of thousands of threads concurrently and offers arithmetic and memory throughput far beyond that of a CPU, but only to programs that fit its restrictive execution model. Writing such programs directly, in low-level programming models such as CUDA or OpenCL, requires detailed knowledge of the hardware and is time-consuming and error-prone.

Data-parallel functional languages offer a more attractive programming model for GPUs. The programmer expresses an algorithm through bulk operations such as map, reduce, and scan, and a compiler is responsible for mapping the exposed parallelism onto the hardware. In such languages, parallelism composes naturally, since a parallel operation is often applied inside another parallel operation, for example a parallel reduction inside a map over the rows of a matrix. This is *nested* data parallelism, pioneered by Blelloch's NESL [1], and it is the natural way to express many algorithms, including irregular ones such as sparse-matrix and graph computations. GPUs, however, only support *flat* parallelism efficiently. A kernel launch creates a fixed hierarchy of threads, and a thread cannot cheaply spawn further parallel work. Bridging this gap is the task of the *flattening transformation*, due to Blelloch and Sabot [2, 3], which converts arbitrarily nested parallelism into flat parallel operations over a representation based on flat data arrays and segment metadata.

Flattening handles arbitrary, irregular nesting, and provably preserves the asymptotic work and span of the source program. Yet it has seen little use in production compilers, because of its cost. Flattening materialises many intermediate arrays and so increases memory traffic, replicates values across flattened iteration spaces, decomposes programs into long chains of memory-bound bulk operations, and dissolves the regular structure that GPU optimisations such as tiling depend on. It also exploits all of the parallelism in a program, which may be far more than the hardware can use. Largely for these reasons, production compilers have avoided full flattening.

Futhark [4] is a statically typed, pure functional data-parallel language designed for efficient compilation to GPUs. Its compiler is mature and generates code competitive

with hand-written GPU kernels, but its treatment of nested parallelism is deliberately incomplete. The compiler's moderate and incremental flattening [4, 5] handle only *regular* nested parallelism, where the width of an inner parallel operation is invariant across the iterations of the enclosing map. When the inner width varies, the parallelism is *irregular*, and the compiler must sequentialise it, leaving the parallelism unexploited. For programs whose parallelism is mostly irregular, this can make the natural formulation unusably slow, and the programmer is forced to flatten the program by hand. Worse, sequentialisation is not always possible. When the shape of an intermediate value is discovered only during execution, for instance a loop-carried array that grows from iteration to iteration, the GPU backend rejects the program outright, so the limitation is then not one of performance but of expressiveness.

This thesis develops a *full flattening* pass for the Futhark compiler that addresses this limitation. The pass implements Blelloch-style flattening of irregular nested parallelism, drawing inspiration from Oancea's work on practical flattening transformations [6], and translates the Futhark nested intermediate representation into the flat GPU representation. It continues the work of Sevald-Krause [7], who prototyped full flattening for a subset of the language. With this pass, the Futhark compiler can for the first time exploit irregular nested parallelism, which can make programs substantially faster whenever exploiting that parallelism is the better choice.

Making full flattening usable in practice, rather than merely correct, is the central concern of the thesis. Classical full flattening is known to incur the costs described above, and a naive implementation would pay them everywhere. Our central idea is that the price of irregularity should be paid only where the program is actually irregular. We detect the *uniformity* of values with respect to the surrounding map nest, that is, whether their shapes are invariant across the iterations of the nest, cheaply and syntactically through Futhark's size-dependent types. When an operation is uniform, we can generate more efficient code. For example, uniform map nests are kept as regular multidimensional operations rather than dissolved into segment metadata. We also avoid unnecessary work by not distributing scalar computations and by leaving the replication of variant free variables implicit rather than materialising it. Finally, because exploiting all parallelism is not always profitable on finite hardware, the pass integrates with incremental flattening [5], generating several semantically equivalent versions of each parallel nest and deferring the choice among them to run time.

1.1 Contributions and Thesis Outline

The central contribution of this thesis is the design of a full flattening transformation that mitigates the classical downsides of flattening. Around this idea, the thesis makes the following contributions.

- We design a full flattening transformation, together with a set of refinements that mitigate the classical costs of flattening. The transformation applies different flattening rules depending on whether a construct is uniform with respect to the surrounding nest or not, which results in more efficient code for uniform

constructs. Variant free variables are replicated only implicitly, and scalar computation is left undistributed (Chapters 3, 4, and 5).

- We implement the transformation as a pass in the Futhark compiler that covers the full Futhark language, and integrate it with incremental flattening (Chapter 6).
- As a result, irregular nested parallelism can now be exploited in Futhark, while the compiler retains its existing ability to compile regular parallelism efficiently. The GPU backend also accepts programs it would previously have rejected (Chapters 6 and 7).
- We evaluate the pass and show that regular workloads are mostly unaffected, while the natural nested formulation of an irregular workload such as sparse-matrix–vector multiplication becomes orders of magnitude faster (Chapter 7).

The remainder of the thesis is organised as follows. Chapter 2 provides background on GPU hardware, data-parallel programming, the flattening transformation, the Futhark language and compiler, and related work. Chapter 3 presents our full flattening transformation on a small language, and Chapter 4 shows how its costs can often be avoided. Chapter 5 extends the transformation to further language constructs. Chapter 6 describes the implementation of the pass in the Futhark compiler, including its integration with incremental flattening, and Chapter 7 evaluates its correctness, expressiveness, and performance. Finally, Chapter 8 concludes and outlines future work.

Chapter 2

Background and Related Work

This chapter provides the background necessary for the rest of the thesis. We first describe the hardware trends that motivate data-parallel programming, together with the execution and memory models of modern GPUs (Section 2.1). We then introduce data-parallel programming and the main concepts used throughout the thesis (Section 2.2). Next, we present the classical flattening transformation and discuss its main costs (Section 2.3). We then describe the Futhark language and compiler, with emphasis on the features and compiler stages on which this thesis builds (Section 2.4). Finally, we survey related work on flattening and the compilation of nested data parallelism (Section 2.5).

2.1 The Rise of Parallel Hardware

For decades, improvements in application performance came mainly from making individual processor cores faster. Each new processor generation increased the speed of sequential programs through higher clock frequencies. This trend largely ended in the early 2000s. Energy consumption and the difficulty of removing the generated heat limit how much the clock frequency can be increased, and how much useful work a single core can perform per clock period. Since then, additional transistors have instead been spent on parallelism by adding more cores, wider vector units, and more hardware threads [8].

Consequently, modern hardware provides more computational resources, but these resources are mostly available as parallel execution capacity rather than as a faster single core. Programs must therefore contain enough parallel work to make effective use of new hardware and the burden of finding and expressing this parallelism has shifted to software.

GPUs are massively parallel processors that are now commonly used for many non-graphics workloads. Compared to CPUs, GPUs follow a different design philosophy. CPUs are optimised for low latency. They spend much of their area and power budget on features such as large caches, branch prediction, and out-of-order execution, which help a single instruction stream run quickly. GPUs are instead optimised for throughput. They dedicate more of their resources to arithmetic units and memory

bandwidth, with the goal of completing a large amount of work in parallel.

This design makes GPUs well suited to computations that can be expressed as many mostly independent and fine-grained tasks. For such workloads, GPUs can provide much higher arithmetic and memory throughput than CPUs, which has motivated the use of GPUs for the computationally intensive parts of many applications.

However, this performance comes with a restrictive programming model. GPUs can be programmed directly using programming models such as CUDA, HIP, or OpenCL, but obtaining good performance requires detailed knowledge of the hardware. In particular, the hardware rewards programs with large numbers of similar, independent, fine-grained tasks and predictable memory access patterns. Programs that do not fit this structure can be difficult to implement efficiently. The rest of this section describes this execution model and GPU memory hierarchy in more detail.

2.1.1 GPU Execution Model

A GPU-accelerated program consists of *host* code, which runs on the CPU, and *device* code, which runs on the GPU. The device code is organised into *kernels*, which are functions executed in a data-parallel manner by a large number of threads. Execution begins on the host when it launches a kernel. This creates a *grid* of threads on the device, all executing the same kernel function. When all threads in the grid have terminated, the kernel finishes, and execution continues on the host until another kernel is launched [8].

The threads of a grid are organised in a two-level hierarchy. A grid consists of a number of *thread blocks*, called *workgroups* in OpenCL terminology. Each block contains the same number of threads, typically a few hundred. Threads within the same block are guaranteed to execute on the same *streaming multiprocessor* (SM). They can communicate through fast on-chip memory and can synchronise using barrier instructions. Threads in different blocks, however, cannot be assumed to run concurrently. Blocks are scheduled onto SMs in an arbitrary order, possibly sequentially, depending on the capacity of the hardware. Consequently, there is no general synchronisation mechanism between blocks within a single kernel.

Within a block, threads are further grouped into *warps* (typically 32 threads), which are the unit of scheduling. The threads of a warp execute in a *single-instruction, multiple-thread* (SIMT) fashion. They share an instruction stream, and at each step the warp executes one instruction on behalf of all its threads.

GPU programming models such as CUDA, HIP, and OpenCL expose this execution model to the programmer. They allow programmers to write code that runs on the GPU, but they also make details visible that are absent from ordinary CPU programming, such as explicit kernel launches, separate host and device code, and explicit data movement between CPU and GPU memory.

A consequence of this execution model is important for this thesis. The parallelism a kernel exposes is *flat*. A kernel launch creates a fixed two-level hierarchy of blocks and threads, and an individual thread cannot cheaply spawn further fine-grained parallel

work.[1] Many programs, however, are naturally expressed with *nested* parallelism, where a parallel operation is applied to each element of a collection and that operation is itself parallel. Such a program cannot be mapped directly onto the hardware. The nested structure must be *flattened* into a single flat level of parallelism, either by the programmer or by a compiler. We make these notions precise in Section 2.2. Here we note only that the hardware provides flat parallelism, whereas programs often express nested parallelism.

2.1.2 GPU Memory Hierarchy

GPU performance is more often limited by memory access than by arithmetic, so the memory system deserves separate attention. A GPU exposes a hierarchy of memory spaces [8]:

- *Registers* are private to each thread and provide the fastest storage available. They are usually used for holding scalar values. The number of registers is limited. A kernel that uses many registers per thread can reduce the number of threads that can be resident on an SM at the same time, which may reduce the GPU's ability to hide memory latency.
- *Shared memory* (*local memory* in OpenCL terminology) is a small on-chip memory, on the order of tens to a couple of hundred kilobytes per SM, shared by the threads of a block. Its latency is close to that of registers and it is much faster than global memory. Shared memory is the primary mechanism for communication and data reuse within a block. Unlike a hardware-managed cache, shared memory is explicitly managed by the program.
- *Global memory* is the large off-chip DRAM, on the order of gigabytes, accessible by all threads and by the host. It has high bandwidth but also high latency and it is slow to access. All communication between blocks, and between kernels, goes through global memory. Thus, global memory is the only way to pass initial data to a kernel, and for a kernel to return a result. Data is transferred between global memory and the host through API calls, at much lower bandwidth than the GPU's own accesses. Global memory is cached, but the caches are small relative to the number of threads, so GPU code cannot rely on caching in the way CPU code can.

Rather than relying on caches, the GPU hides memory latency through massive multithreading. When a warp stalls on a memory access, the SM simply switches to another resident warp. This works only if enough warps are resident, which again ties efficiency to the amount of available parallelism.

The pattern of global memory accesses matters as much as their number. The memory system services a warp's accesses in units of large contiguous transactions, so

[1] CUDA supports *dynamic parallelism*, where a kernel can launch other kernels, but the overhead is substantial, and it is rarely an efficient way to express fine-grained nested parallelism.

when the threads of a warp access adjacent addresses in the same cycle, the accesses *coalesce* into few transactions. Scattered accesses, by contrast, waste most of each transaction's bandwidth. Efficient GPU code must therefore expose regular, analysable access patterns that can be mapped to coalesced global-memory accesses, or that allow data to be staged through shared memory and reused across the threads of a block.

This is one reason why optimisations such as tiling are especially important on GPUs. If a computation repeatedly accesses the same data, it can be much faster to load a tile into shared memory once and then reuse it across the threads of a block. The structure of the program therefore matters not only for exposing enough parallelism, but also for making memory accesses regular, coalesced, and amenable to reuse.

These properties of the memory system also constrain what device code may do. Because registers and per-thread stack space are scarce, each thread can keep only a small amount of state, which together with the lockstep execution of warps makes recursion and deep call chains impractical. In the execution model Futhark targets, allocation is a host-side operation, so the memory a kernel needs is allocated before it is launched rather than grown during its execution. And since the host and device have separate memory spaces, data must be explicitly copied between them before and after a kernel runs.

2.2 Data Parallelism

In the data-parallel programming model, parallelism is expressed by applying the same operation to all elements of a collection at once, rather than by managing synchronisation explicitly [9]. In a functional setting this takes the form of *second-order array combinators* (SOACs) such as map, reduce, and scan, which take a function as an argument and apply it across an array.

This model is attractive because it lets the programmer express an algorithm without reasoning about threads, synchronisation, memory management, or race conditions. The semantics is sequential, which makes programs easy to reason about, yet the same bulk operations expose large amounts of fine-grained parallelism that a compiler can exploit. The model therefore offers correctness, performance, and ease of programming at the same time [10].

2.2.1 Flat and nested parallelism

We distinguish between two types of parallelism:

Definition 2.1: Flat and nested parallelism

Data parallelism is flat when the function applied by a parallel combinator is itself sequential, and nested when that function may itself contain parallel operations.

A simple example of flat parallelism is:

```
map (\x -> x + 1) xs
```

Here, the function `\x -> x + 1` performs no further parallel work. Flat parallelism maps straightforwardly onto a GPU kernel, with one array element per thread, but it is a poor fit for how programmers naturally structure algorithms. Many computations are more naturally written by applying a parallel operation inside another parallel operation. For example, the following expression sums each row of a two-dimensional array:

```
map (\row -> reduce (+) 0 row) xss
```

Here, the outer map is parallel over the rows, and its body is itself a parallel reduce. This is nested data parallelism. It arises naturally in many settings. A matrix computation maps a parallel operation over rows, a graph algorithm maps over vertices and processes the edges of each vertex in parallel, and a divide-and-conquer algorithm recursively spawns parallel subproblems. Nested data parallelism was pioneered by Blelloch's NESL [1, 11], which demonstrated that many irregular parallel algorithms have concise nested data-parallel formulations.

Nested parallelism is exactly the structure that GPU hardware does not support directly (Section 2.1.1). One option is to exploit only the outermost level of parallelism, but this may not expose enough parallelism to saturate the hardware. Exploiting the inner parallelism on a GPU is itself difficult, and is the concern of this thesis.

2.2.2 Regular and irregular parallelism

Nested parallelism divides further into two categories:

Definition 2.2: Regular and irregular parallelism

An inner parallel operation is regular with respect to an enclosing map *if its amount of work (its width) is invariant across the iterations of that* map*, and irregular otherwise.*

Consider these two programs:

```
map (\i -> reduce (+) 0 (0...4)) (1...n)
map (\i -> reduce (+) 0 (0...i)) (1...n)
```

The first is regular. Every one of the n iterations does the same amount of inner work, namely a reduction over the five elements of `0...4`, so the combined iteration space is a dense $n \times 5$ rectangle.

The second program is irregular, because the inner reduction now runs over `0...i`, whose length depends on the current iteration. The amount of inner work therefore varies from one iteration to the next, and the combined iteration space is ragged rather than rectangular.

2.2.3 Uniformity

The regular and irregular distinction can be characterised in terms of the *uniformity* of arrays relative to a surrounding map nest. We first distinguish between variant and invariant values:

Definition 2.3: Value variation and array uniformity

A value is variant with respect to a map nest if it may take a different value in different iterations of that nest, and invariant otherwise. An array is uniform with respect to a map nest if its shape is invariant across all iterations of the nest, and non-uniform when its shape depends on a variant value.

Consider the following program:

```
map (\i -> 0...4) is
```

It produces an array of length 5 in every iteration, so its intermediate arrays are uniform.

An array is non-uniform when its shape depends on a variant value, in which case different iterations may produce arrays of different lengths. For example, consider the following program:

```
map (\i ->
       let intermediate = 0...i
       in reduce (+) 0 intermediate)
    is
```

The outer map returns one scalar per iteration, but the intermediate array is non-uniform because its length depends on the current value of i. The inner reduction therefore processes a different width in different outer iterations.

Uniformity and regularity are closely related. Non-uniformity is exactly what gives rise to irregular parallelism. If the source of an inner parallel operation is a non-uniform array, that parallelism is irregular. A map over a non-uniform array, for instance, is an irregular map, since its iterations range over inner arrays of differing lengths. The same effect arises through control flow. A conditional or loop whose body contains inner parallelism produces irregular parallelism when the branch condition is variant in the surrounding nest, or when the trip count of the loop is variant, since the amount of inner parallelism differs from one iteration of outer surrounding map to the next.

We note that if the shape of every array is recorded, uniformity can be decided syntactically by inspecting the values that make up an array's shape. If those values are bound inside the surrounding map nest, they may be variant in the nest, and we conservatively treat the array as non-uniform. If they are bound outside the nest, they are invariant, and the array is uniform. Because of the relationship between uniformity and regularity, this gives a cheap and conservative way to detect irregular parallelism syntactically.

2.2.4 The Work–Span Cost Model

To reason about the parallelism of a program independently of any particular machine, we use the *work–span* model as our cost model for data-parallel programs:

Definition 2.4: Work and span

The work of a program, denoted W, is the total number of operations it performs. The span, also called depth and denoted D, is the length of the longest chain of dependent computations. It corresponds to the cost of evaluating the computation on an ideal machine with infinite processors, where all independent operations run simultaneously.

For example, consider the following nested parallel program:

```
map (\x -> map (\n -> x * n) ns) xs
```

Assume that `xs` has length n and `ns` has length m. The program performs one multiplication for each pair of elements drawn from `xs` and `ns`, so its work is $O(nm)$. Since all iterations of both maps are independent, the span is $O(1)$.

Now consider a version in which the inner parallel map is replaced by a sequential loop:

```
map (\x ->
       loop acc = replicate m 0
       for i < m do
         acc with [i] = x * ns[i])
    xs
```

This program is semantically equivalent to the nested-map version, and it still performs $O(nm)$ work, since each pair of elements is multiplied exactly once and each accumulator update is performed in place with $O(1)$ cost. Its span, however, is now $O(m)$, because each outer iteration must execute the inner loop sequentially. The outer map still exposes parallelism across the n elements of `xs`, but the parallelism over `ns` has been lost.

We say that a program transformation is *work-preserving* if it does not asymptotically increase W, and *depth-preserving* if it does not asymptotically increase D. In a data-parallel setting, it is desirable for transformations to preserve both properties, since work preservation ensures that the transformed program does not perform asymptotically more computation, while depth preservation ensures that the transformation does not remove asymptotic opportunities for parallel execution.

2.3 The Flattening Transformation

As we have seen, nested parallelism arises naturally when writing in a data-parallel language, but GPUs and other parallel hardware efficiently support only flat parallelism and struggle to execute nested parallelism directly. It is therefore useful to have a

transformation that converts nested parallelism into the flat parallelism that GPUs run well. This section gives a high-level background of the flattening transformation. Later, in Chapter 3, we describe our flattening transformation with examples.

The classic technique is the *flattening transformation*, also called *vectorisation*, due to Blelloch and Sabot [2, 3]. The key idea is to represent a nested array by a flat data array together with *segment metadata*, which in the simplest form is a descriptor recording the length of each inner array, and to replace every parallel operation nested inside a map by a *segmented* operation that processes all segments at once on the flat representation. For example, a reduction inside a map is turned into a single segmented reduction that reduces every inner array of `xss` in one flat parallel operation, rather than running one reduction per outer iteration. Segmented operations can themselves be implemented using a small set of flat primitives, notably scans, which Blelloch showed to be an efficient and versatile parallel primitive [12]. The transformation therefore provides a systematic way to compile arbitrarily nested parallelism into flat parallel operations.

Flattening has a strong theoretical foundation. Under the restrictions and cost model of the classical analysis, nested data-parallel programs can be transformed into flat data-parallel programs while preserving asymptotic work and span [3]. NESL demonstrated this approach in practice by compiling nested parallel programs to VCODE, a flat vector intermediate language.

Classical *full flattening*, however, has well-known costs, which explain why it has seen little adoption in production compilers. In essence, the transformation trades locality for parallelism. Intermediate values that could previously stay in registers become materialised arrays after flattening, which increases memory traffic. In particular, a value that was a scalar in every iteration may have to be replicated across the flattened iteration space. This matters especially on GPUs, where such values might otherwise reside in registers, but after flattening may become materialised arrays that increase global memory traffic. Moreover, flattening dissolves the regular structure of any uniform sub-computation into segment metadata, which forecloses locality optimisations such as tiling that depend on that structure. This is especially damaging on GPUs, where, as discussed in Section 2.1.2, there is little cache and tiling optimisation is essential for high performance. Finally, since real hardware has only finite parallel resources, exploiting all the parallelism in a program is not always beneficial, and a fully flattened program can be slower than one that leaves some parallelism unexploited. Much of this thesis is concerned with recovering, within a flattening-based compiler, the structure and locality that naive full flattening destroys.

2.4 Futhark

Futhark is a statically typed, pure functional, data-parallel array language designed for efficient compilation to GPUs [4]. Syntactically it resembles a small ML-family language. Semantically it is a nested data-parallel language in the tradition of NESL, but with a design deliberately constrained to permit aggressive compiler optimisation. The main goal of Futhark is to allow programmers to take advantage of parallel

hardware, such as GPUs, without requiring detailed knowledge of the low-level details of such hardware. This makes it easier for users to express their algorithms in Futhark than in low-level GPU programming languages such as CUDA. The work in this thesis is implemented as a pass in the Futhark compiler, so we describe the language and compiler in some detail.

2.4.1 The Language

Parallelism in Futhark is expressed through a small set of built-in array combinators. The most important of these are Second-Order Array Combinators (SOACs), such as map, scan, reduce and hist:

- map $f\ xs$ applies f to every element of xs. Variants such as map2 apply a function elementwise to several arrays at once.
- reduce $\oplus\ z\ xs$ combines the elements of xs with the operator $\oplus$, which must be associative with neutral element z, so that the compiler may evaluate the reduction in parallel.
- scan $\oplus\ z\ xs$ computes all prefix reductions of xs, under the same requirements on $\oplus$.
- hist $\oplus\ ne\ k\ is\ vs$ computes a generalised histogram: it produces an array of length k in which, for each position j, the value $vs[j]$ has been combined into position $is[j]$ using the operator $\oplus$, starting from the neutral element ne. $\oplus$ must be associative and commutative.

In addition to SOACs, Futhark provides other built-in array operations such as replicate, iota, reshape, and transpose. These operations also contribute to the structured array-programming style of Futhark, and many of them can be implemented efficiently in parallel. Together, SOACs and these array operations provide the main way of expressing data-parallel computation in Futhark programs.

As a deliberate design decision, Futhark is not a general-purpose language. It omits features such as recursive data types and function recursion, in exchange for a compiler that can produce code competitive with hand-written GPU kernels [4]. Futhark does, however, provide sequential loop constructs that are semantically equivalent to tail-recursive functions.

The following Futhark program, which computes matrix-vector multiplication, illustrates programming in Futhark:

```
def dotprod [m] (xs: [m]f32) (ys: [m]f32) : f32 =
  reduce (+) 0 (map2 (*) xs ys)

def matvec [n][m] (A: [n][m]f32) (v: [m]f32) : [n]f32 =
  map (\row -> dotprod row v) A
```

The program exhibits regular nested parallelism, namely a map over the n rows of A, each iteration of which contains m-element map2 and reduce operations.

2.4.2 Size-Dependent Array Types

A distinctive feature of Futhark, used throughout in this thesis, is that array sizes are part of the type system [13]. The type $[n]$f32 denotes an array of exactly n 32-bit floats, where n is a term-level variable of type i64. Functions may bind *size parameters*, as $[n][m]$ above, which are implicit arguments instantiated at each call site. The type of matvec thus statically expresses that the vector length must match the inner dimension of the matrix, and that the result has one element per matrix row. Operations whose result size cannot be expressed in terms of the input sizes, such as filter, are given *existential* sizes, which the type system tracks and the compiler materialises as ordinary variables.

Futhark also requires that all source-level arrays be *regular*, meaning that the inner arrays of a multidimensional array all have the same shape. This is a deliberate design decision. Regular arrays admit a simple flat memory layout and predictable performance, which is central to Futhark's goal of generating efficient GPU code. Size types are the mechanism that enforces this choice. An irregular array has no single size that could be written in its type, so a program such as map iota *ns*, whose result would be irregular, is rejected by the type checker.

For our purposes, the most useful aspect of size types is that they give the compiler a cheap, syntactic handle on the uniformity analysis that drives flattening. An inner parallel operation is regular with respect to an enclosing map exactly when its size expressions are invariant to that map, which the compiler can decide by inspecting whether the relevant sizes are bound inside or outside the map lambda.

2.4.3 The Compiler Pipeline

The Futhark compiler has a conventional architecture, divided into a frontend, a middle-end, and a backend [14].

The *frontend* parses and type-checks the source program, and transforms it into the SOACS dialect of the core intermediate representation. The core IR has no modules, is monomorphic and first-order, and does not have tuples or records. Arrays of tuples are instead turned into multiple arrays. More importantly, the core IR is in A-normal form [15], so every intermediate value is let-bound and expressions are flat. It also retains the size types of the source language, and every array type in the IR carries an explicit size expression.

The *middle-end* is organised as *passes*, which are pure functions from programs to programs, composed into *pipelines* that differ per compiler backend. All pipelines start from the SOACS representation. At this level, the compiler performs several optimisations, most importantly *fusion*, which combines producer and consumer SOACs to eliminate intermediate arrays [10]. In the dotprod function above, fusion combines the map2 and the reduce into a single pass over the input, so the elementwise products are never stored in memory.

Fusion also shapes how SOACs are represented in the core IR, which matters for this thesis because the flattening pass consumes the IR after fusion has run. Rather than representing map, reduce, and scan as separate constructs, the core IR has

a single construct, the Screma, which combines a number of scans, a number of reductions, and a map into one operation, with the plain SOACs as special cases. The fused forms the compiler recognises are the redomap, a reduction applied to the results of a map, the scanomap, a scan applied to the results of a map, and the maposcanomap, which additionally applies a map to the results of the scan.

For GPU compilation, the program must then be transformed so that it contains no nested parallelism. The result is expressed in the GPU IR, in which parallelism takes the form of flat *segmented operations*, such as segmented maps, reductions, and scans over explicit multidimensional iteration spaces. Each segmented operation corresponds roughly to one GPU kernel.

In the existing Futhark compiler, the SOACS to GPU IR transformation is performed by the `ExtractKernels` pass. The pass developed in this thesis is an alternative to the `ExtractKernels` pass. It accepts SOACS IR as input, performs flattening and produces the GPU IR as output.

After this pass, other middle-end passes add memory information and perform further optimisations. The program is then translated to an imperative representation called ImpCode, from which the code generators emit CUDA, HIP, or OpenCL code, together with host code that manages buffers and kernel launches. Because the pass developed in this thesis produces ordinary GPU programs, all of these later stages, as well as the GPU-level optimisations between them, apply unchanged to its output.

2.5 Related Work

NESL and its descendants. The flattening transformation originates with Blelloch and Sabot [2] and was realised in NESL [1, 11], which compiles nested data-parallel programs to the flat vector language VCODE. NESL predates GPUs, but its model has been ported to them. Bergstrom and Reppy [16] adapted the NESL compiler to target CUDA, executing the generated code with a modified version of the VCODE interpreter and their own CUDA implementation of the underlying vector library. Because their approach fully flattens the program and VCODE lacks the information needed for optimisations such as fusion, the resulting code suffers from high memory traffic, and performance is limited in practice. CuNesl [17] compiles NESL programs to the GPU and maps them to different levels of parallelism, attempting to preserve some of the nesting structure rather than flattening it completely. It did not focus on kernel fusion, but rather on implementing recursive calls efficiently. Nessie [18] is another NESL to CUDA compiler that supports the optimisations the earlier approach did not, in particular fusion, which it enables through a shape analysis on the flattened code.

Data Parallel Haskell. Data Parallel Haskell (DPH) [19] is an extension of Haskell that brings nested data parallelism into a general-purpose language. Whereas NESL is a small first-order language, DPH applies the flattening transformation in a much richer setting, extending it to higher-order functions, user-defined algebraic data types, and a polymorphic type system. DPH targets multicore CPUs rather than GPUs, and

is implemented inside the Glasgow Haskell Compiler. Of particular relevance to this thesis is the vectorisation-avoidance optimisation of Keller et al. [20], which keeps scalar computations unvectorised inside parallel contexts instead of lifting them to arrays, and which corresponds closely to the treatment of scalar statement groups during distribution in this thesis.

Data-only flattening. Data-only flattening [21] is another approach, implemented in the Manticore compiler for Parallel ML. Instead of fully flattening a program, which converts both its nested data structures and its nested control structures into flat form, data-only flattening transforms only the data representation and leaves the control structure intact. Flattening the data gives the benefits of a flat representation, while keeping the original control structure avoids some of the costs of full flattening. This is suitable for multicore CPUs, where the runtime can execute the nested parallel control directly. The technique, however, does not transfer to GPUs, since, as discussed in Section 2.1.1, a GPU cannot execute nested parallel control and therefore requires the control structure to be flattened as well.

Flattening in Futhark. The first approach to compiling nested parallelism in Futhark was *moderate flattening* [4]. It is a set of rewrite rules, such as map fission and map–loop interchange, that reorganise regular nests of SOACs into perfect map nests, which then become multidimensional SegOps. Moderate flattening exploits only parallelism that is regular, and only up to a practical depth. Inner parallelism that cannot be profitably mapped to the hardware is turned into sequential loops inside the generated kernels. Sequentialising inner parallelism is often desirable, as it preserves locality, but moderate flattening makes this choice statically and heuristically.

Incremental flattening [5] later refined this approach, and is the strategy the compiler uses today. Rather than committing to one flattening at compile time, it generates multiple semantically equivalent flattenings of each nest: one version that parallelises only the outer levels and sequentialises the rest, one that maps an inner level to the threads of a block and uses shared memory (an *intra-block*, or intra-group, kernel), and one that fully parallelises across the grid. The versions are guarded by predicates comparing the dynamic sizes of the parallel dimensions against tunable thresholds, so the choice among them is deferred to run time, and the thresholds can be autotuned per hardware and workload [5].

Crucially, both moderate and incremental flattening handle only regular nested parallelism. When the size of an inner parallel operation varies across the iterations of an enclosing map, neither technique applies, and the compiler falls back to sequentialising the inner operation entirely, where this is possible. For programs whose parallelism is irregular, such as many sparse and graph computations, this can leave the bulk of the available parallelism unexploited, and the programmer is left to flatten the program by hand. Manual flattening is effective but laborious and error-prone. The *flattening-by-expansion* technique [22] packages some of the recurring patterns as a library, but the burden of reasoning about segment metadata still falls on the programmer.

Sequentialisation is not always available as a fallback, however, and in such cases the limitation is not merely one of performance but of expressiveness as some programs are rejected outright. The difficulty arises when an intermediate array has a shape that varies across a parallel construct, and in particular when that shape is not known on entry to the construct but is produced inside it. A loop-carried array that grows from iteration to iteration is one such case:

```
map (\n ->
       let res =
         loop arr = [1]
         while length arr < n do
           arr ++ arr
       in i32.sum res)
    ns
```

This program type-checks, but the GPU backend rejects it. Each invocation of the `map` produces a `res` whose length is discovered only as the loop runs, rather than being known on entry to the `map`, so the compiler cannot pre-compute a per-iteration size with which to lay the results out contiguously. The consequence is not slower code but no GPU-compiled code at all. The programmer must restructure the program by hand or forgo the GPU backend entirely.

Futhark has therefore been developing a full flattening pass that implements Blelloch-style flattening of irregular nested parallelism directly in the compiler. Sevald-Krause's bachelor's thesis [7] prototyped full flattening of irregular nested parallelism for a subset of the language, and the present thesis continues the work. The contributions of this thesis extend this pass along several axes: broadening its coverage from a subset of the language towards the whole of Futhark, detecting and exploiting uniform statements, avoiding unnecessary replication of such values, and integrating with incremental flattening.

Chapter 3

Full Flattening

This chapter presents our full flattening transformation, which is loosely based on Blelloch's seminal work [2] and Oancea's work [6] on flattening that transforms a program with nested parallelism into one in which all parallelism is flat. We begin by defining a small source and target language (Section 3.1). We then give an overview of the full flattening process using a running example (Section 3.2) before presenting the transformation in detail.

The transformation consists of three main steps. First, distribution rewrites a map nest into a sequence of maps, each containing a single statement (Section 3.3). Second, the flattening rules replace these maps with segmented operations over the whole nest (Section 3.4). Finally, the segmented operations and the irregular arrays they operate on are lowered to flat parallel operations over a representation based on flat data arrays and segment metadata (Sections 3.5 and 3.6). We end the chapter by extending the transformation to nested maps with free variables and discussing the resulting replication problem (Section 3.7).

3.1 The Language

We use a small source language and a small target language to describe the flattening transformation. The source language describes programs before flattening. The target language extends the source language with constructs used during and after flattening.

3.1.1 Source Language

The grammar of the source language is shown in Figure 3.1. The source language is a pure functional language with map as a second-order parallel array combinator.

Here, τ denotes types, e expressions, s let-bound statements, b blocks, $f\,d$ function definitions, and p programs. The metavariable v ranges over variables and constants. A bar denotes a sequence, so $\bar{v}$ means a sequence of values. The metavariable op ranges over built-in operations, while f ranges over user-defined functions. The set of built-ins is left open, because we extend it with additional operations introduced by the flattening transformation.

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid [n]\tau \mid []\tau \\
e & ::= & v \mid v \oplus v \mid op\ \bar{v} \mid f\ \bar{v} \mid \mathsf{if}\ v\ \mathsf{then}\ b\ \mathsf{else}\ b \mid \mathsf{iota}\ v \mid \mathsf{sum}\ v \\
 & & \mid \mathsf{rearrange}\langle d_1, \ldots, d_r \rangle\ v \\
 & & \mid \mathsf{fold}\ (\lambda(acc : \tau)\ (x_1 : \tau_1), \ldots, (x_r : \tau_r) \to b)\ v_0\ v_1 \cdots v_r \\
 & & \mid \mathsf{map}\ (\lambda(x_1 : \tau_1), \ldots, (x_n : \tau_n) \to b)\ v_1 \cdots v_n \\
s & ::= & \mathsf{let}\ (x_1 : \tau_1), \ldots, (x_n : \tau_n) = e \\
b & ::= & s_1 \ldots s_m\ \mathsf{in}\ v_1, \ldots, v_n \\
fd & ::= & \mathsf{fun}\ f\ ((x_1 : \tau_1), \ldots, (x_n : \tau_n)) : (\tau'_1, \ldots, \tau'_m) = b \\
p & ::= & fd_1 \ldots fd_k
\end{array}
$$

Figure 3.1: Grammar of the source language.

Expressions have their standard semantics. For example, iota n produces the array $[0, \ldots, n-1]$, and sum xs reduces an array of integers by addition.

The expression rearrange$\langle d_1, \ldots, d_r \rangle$ xs permutes the dimensions of xs. The metavariables $(d_1, \ldots, d_r)$ range over dimension indices and must form a permutation of the dimensions of xs. The operations map, iota, rearrange and sum are bulk operations and may be executed in parallel when possible.

We also include a sequential fold operation. The expression fold $(\lambda(acc : \tau)\ (x_1 : \tau_1), \ldots, (x_r : \tau_r) \to b)\ z\ xs_1 \cdots xs_r$ reduces the arrays $xs_1, \ldots, xs_r$ together, starting from the initial accumulator z. The input arrays must have the same length. At each iteration, the fold function receives the current accumulator and the corresponding element from each input array. Since a fold is evaluated sequentially, we require its result to have a shape that is invariant across the iterations.

The source language is statically typed. Every regular array type carries an explicit size, which may be either a constant or a symbolic size. We write $[n]\tau$ for an array of length n whose elements have type τ.

All arrays in source-language programs are regular. This means that all inner arrays of a multidimensional array have the same shape. For example, the array $[[1, 2], [3, 4]]$ can be represented as a regular source-language array of type $[2][2]$int.

In contrast, $[[1, 2], [3]]$ is an irregular array, because its inner arrays have different lengths. We write the type of this array as $[2][]$int. The notation $[]\tau$ denotes an irregular array of elements of type τ. Although irregular arrays do not occur in source-language programs, they may appear as intermediate values after distribution. We therefore include $[]\tau$ in the type notation used to describe the intermediate programs produced by the transformation.

We also note that, in examples, we may omit type annotations when they are clear from context. We also use the abbreviation let $\bar{y} = e$ in $\bar{y} \equiv e$ when the let-binding is only administrative. Finally, we may omit repeated occurrences of in in sequences of let-bindings.

3.1.2 Target Language

The target language extends the source language with flat parallel constructs and with additional operations used during flattening. Its grammar is shown in Figure 3.2.

$$
\begin{array}{rcl}
e^T & ::= & e \mid op^T\ \bar{v} \mid \mathsf{replicate}\ v\ v \mid \mathsf{scatter}\ v\ v\ v \\
& & \mid \mathsf{segmap}\langle i_1 < n_1, \ldots, i_d < n_d \rangle\ b^T \\
& & \mid \mathsf{segred}\langle i_1 < n_1, \ldots, i_d < n_d \rangle\ \oplus\ z\ b^T \\
& & \mid \mathsf{segscan}\langle i_1 < n_1, \ldots, i_d < n_d \rangle\ \oplus\ z\ b^T \\
s^T & ::= & \mathsf{let}\ (x_1 : \tau_1), \ldots, (x_n : \tau_n) = e^T \\
b^T & ::= & s_1^T \ldots s_m^T\ \mathsf{in}\ v_1, \ldots, v_n \\
fd^T & ::= & \mathsf{fun}\ f\ ((x_1 : \tau_1), \ldots, (x_n : \tau_n)) : (\tau'_1, \ldots, \tau'_m) = b^T \\
p^T & ::= & fd_1^T \ldots fd_k^T
\end{array}
$$

Figure 3.2: Grammar of the target language.

The metavariable op^T ranges over target-level built-in operations. The abstract operations produced by flattening, written segmented_iota, segmented_sum, segmented_rep, and segmented_scan, also appear during the transformation. These are lowered to the target constructs and primitives, and we distinguish them by the segmented_ prefix. All of these operations are introduced when they are first used, and their lowering is described in later sections. The expression replicate $n\ v$ produces an array of length n where every element is v. The expression scatter $dest\ is\ vs$ updates the array $dest$, where is is an array of destination indices and vs is an array of values. For each position j, the value $vs[j]$ is written to position $is[j]$ in the result. We use scatter to describe how flat arrays are constructed or updated during lowering.

The target construct $\mathsf{segmap}\langle i_1 < n_1, \ldots, i_d < n_d \rangle\ b$ denotes a flat parallel map over the regular product space $n_1 \times \cdots \times n_d$. The names $i_1, \ldots, i_d$ are logical indices bound in the body b. If the body has type τ, then the result has type $[n_1] \cdots [n_d]\tau$.

Semantically, segmap produces the same result as the corresponding nest of ordinary maps,

$$
\begin{array}{l}
\mathsf{segmap}\langle i_1 < n_1, \ldots, i_d < n_d \rangle\ b \\
\quad \equiv \mathsf{map}\ (\lambda i_1 \to \cdots \mathsf{map}\ (\lambda i_d \to b)\ (\mathsf{iota}\ n_d) \cdots)\ (\mathsf{iota}\ n_1)
\end{array}
$$

The difference is in execution. The right-hand side describes the meaning using nested source-level maps, while the left-hand side denotes a single target-level flat parallel operation over the product space $n_1 \times \cdots \times n_d$. The target notation also uses explicit logical indices rather than mapping directly over input arrays. This keeps the regular multidimensional structure visible.

The target constructs segred and segscan are the reduction and scan counterparts of segmap. Like segmap, they range over a regular index space, but the innermost index is the dimension along which the values are combined. The construct $\mathsf{segred}\langle i_1 < n_1, \ldots, i_d < n_d \rangle\ \oplus\ z\ b$ reduces the body b along the innermost dimension i_d with an

associative operator $\oplus$ and neutral element z, keeping the outer dimensions, so the result has type $[n_1] \cdots [n_{d-1}]\tau$. The construct $\mathsf{segscan}\langle i_1 < n_1, \ldots, i_d < n_d \rangle \ \oplus \ z \ b$ computes the corresponding scan along i_d, keeping the full shape $[n_1] \cdots [n_d]\tau$. As with segmap, both denote a single flat parallel kernel over the regular index space and introduce no nested parallelism.

It is important to note that segmap, segred, and segscan cannot be nested in the target language. In other words, their bodies may not contain further parallel operations. Any expression that remains in such a body is therefore executed sequentially.

3.2 Overview

Before describing the transformation in detail, we sketch the whole process on the running example, so that the later sections fill in a structure the reader already has in mind.

We use the following expression as a running example:

$$
\begin{array}{l}
e : [k]\mathsf{int} = \\
\quad \mathsf{map}\ (\lambda(i : \mathsf{int}) \rightarrow \\
\qquad \mathsf{let}\ zs : [i]\mathsf{int} = \mathsf{iota}\ i \\
\qquad \mathsf{let}\ res : \mathsf{int} = \mathsf{sum}\ zs \\
\qquad \mathsf{in}\ res)\ is
\end{array}
$$

To get a feeling for how the language works, let us execute the running example on the input $is = [1, 2, 3]$. The outer map has three iterations. Although map is a parallel operator, its meaning can be understood by considering the iterations individually. In the first iteration, $i = 1$, so $\mathsf{iota}\ 1 = [0]$, and therefore $\mathsf{sum}\ [0] = 0$. In the second iteration, $i = 2$, so $\mathsf{iota}\ 2 = [0, 1]$, and therefore $\mathsf{sum}\ [0, 1] = 1$. In the third iteration, $i = 3$, so $\mathsf{iota}\ 3 = [0, 1, 2]$, and therefore $\mathsf{sum}\ [0, 1, 2] = 3$. Thus, the whole expression evaluates to $[0, 1, 3]$.

This example illustrates why flattening is useful. The outer map expresses parallelism across the elements of is, and each iteration can be evaluated independently. Inside each iteration, however, the program performs further bulk array operations, namely $\mathsf{iota}\ i$ and $\mathsf{sum}\ zs$. These operations expose additional data parallelism inside the body of the outer map. Therefore, the program contains nested parallelism. Flattening transforms this nested structure into flat data-parallel operations that are better suited to execution on GPUs.

Flattening removes this nested structure in three steps. First, distribution separates the statements in the map body:

$$
\begin{array}{l}
\mathsf{let}\ zss = \mathsf{map}\ (\lambda i \rightarrow \mathsf{iota}\ i)\ is \\
\mathsf{let}\ ress = \mathsf{map}\ (\lambda zs \rightarrow \mathsf{sum}\ zs)\ zss \\
\mathsf{in}\ ress
\end{array}
$$

Second, the flattening rules replace these maps with segmented operations:

$$
\begin{array}{l}
\text{let } zss : [k][]\text{int} = \text{segmented\_iota } is \\
\text{let } ress : [k]\text{int} = \text{segmented\_sum } zss \\
\text{in } ress
\end{array}
$$

Finally, lowering represents the irregular values using flat data and segment metadata, and implements the segmented operations using flat parallel primitives. The result is a program with only regular arrays and flat parallel operations:

$$
\begin{array}{l}
\text{let } zss_S : [k]\text{int} = is \\
\text{let } zss_O : [k]\text{int} = \text{scan}_{exc}\ (+)\ 0\ zss_S \\
\text{let } m : \text{int} = \text{sum } zss_S \\
\text{let } zss_F : [m]\text{bool} = \text{gen\_flags } m\ zss_S\ zss_O \\
\text{let } ones : [m]\text{int} = \text{replicate } m\ 1 \\
\text{let } zss_D : [m]\text{int} = \text{segPrefixSum}_{exc}\ zss_F\ ones \\
\text{let } scan\_res\_D : [m]\text{int} = \text{segPrefixSum } zss_F\ zss_D \\
\text{let } red\_res\_D : [k]\text{int} = \text{segmap}\langle j < k \rangle \\
\quad \text{let } s = zss_S[j] \\
\quad \text{let } o = zss_O[j] \\
\quad \text{in if } s = 0 \text{ then } 0 \text{ else } scan\_res\_D[o + s - 1] \\
\text{in } red\_res\_D
\end{array}
$$

3.3 Distribution

The first step of the flattening transformation is distribution. Distribution is essentially map fission. A map whose body contains several let-bound statements is rewritten into a sequence of maps, each of which contains a single statement. This gives the flattening rules a simple normal form to operate on.

At this stage, we assume that maps do not have any free variables. We address this limitation in Section 3.7.

Consider a map $\text{map } (\lambda \bar{x} \to b)\ \bar{xs}$, where $\bar{x}$ are the lambda parameters and $\bar{xs}$ are the arrays being mapped over.

Suppose the body block begins with a let-bound statement:

$$
\begin{array}{l}
\text{let } \bar{y} = e \\
b'
\end{array}
$$

where b' is the remaining block. Distribution rewrites the map as:

$$
\begin{array}{l}
\text{map } (\lambda \bar{x} \to \text{let } \bar{y} = e\ b')\ \bar{xs} \\
\rightsquigarrow \\
\text{let } \bar{ys} = \text{map } (\lambda \bar{x} \to \text{let } \bar{y} = e \text{ in } \bar{y})\ \bar{xs} \\
\text{in map } (\lambda \bar{x}\ \bar{y} \to b')\ \bar{xs}\ \bar{ys}
\end{array}
$$

In the general case, the second map receives both the original inputs $\bar{xs}$ and the intermediate results $\bar{ys}$, since the remaining body b' may depend on both. Distribution repeats this transformation until every map body contains only a single let-bound expression.

In the running example, distribution separates the two statements in the body. A statement inside the body describes the computation performed by one iteration of the original map. After lifting, this statement is turned into a new map that performs the computation for all iterations. First, the iota statement is lifted into its own map:

$$\mathsf{let}\ zss = \mathsf{map}\ (\lambda(i : \mathsf{int}) \to \mathsf{iota}\ i)\ is$$

where the name zss denotes the collection of all per-iteration values of zs. The remaining statement is then mapped over the intermediate result:

$$\mathsf{let}\ ress = \mathsf{map}\ (\lambda(zs : []\mathsf{int}) \to \mathsf{sum}\ zs)\ zss$$

Thus, the distributed form of the running example is:

$$\begin{aligned}&\mathsf{let}\ zss : [k][]\mathsf{int} = \mathsf{map}\ (\lambda(i : \mathsf{int}) \to \mathsf{iota}\ i)\ is\\&\mathsf{let}\ ress : [k]\mathsf{int} = \mathsf{map}\ (\lambda(zs : []\mathsf{int}) \to \mathsf{sum}\ zs)\ zss\\&\mathsf{in}\ ress\end{aligned}$$

In each individual source-level iteration, iota i produces a regular array of type $[i]$int. However, after distribution, the results of all iterations are collected. Since i is variant in the map nest, it may have a different value in each iteration. Therefore, the arrays collected in zss may have different lengths. In other words, zs has a variant shape, so it is non-uniform with respect to the surrounding map. Consequently, the distributed value zss is an irregular array. We write its intermediate type as $[k][]$int.

The final value $ress$, on the other hand, has type $[k]$int and is regular, because each segment of zss produces exactly one integer.

This illustrates the relation between uniformity and regularity after distribution. If an array produced inside a map nest is uniform, then distribution collects the per-iteration results into a regular array. If it is non-uniform, then distribution collects the per-iteration results into an irregular array, as in the case of zss.

3.4 Flattening Rules

After distribution, we have a map nest with a single expression. The flattening rules operate on such map nests. Intuitively, they replace an operation applied independently at the innermost level of the nest by a segmented operation over the whole nest.

3.4.1 Flattening Iota

Consider a map nest whose innermost expression is an iota:

$$\mathsf{map}\ (\lambda x_1 \to \cdots \mathsf{map}\ (\lambda x_p \to \mathsf{iota}\ n)\ xs_p \cdots)\ xs_1$$

Here, n is a scalar value at the innermost level of the map nest. Across the whole map nest, these scalar values form an array. Flattening replaces the nested applications of iota by a segmented_iota expression, which is a segmented iota. The abstract meaning of segmented_iota is that it applies iota to each scalar element of its input, while preserving the outer structure. For example:

$$\text{segmented\_iota } [1,2,3] = [[0],[0,1],[0,1,2]]$$

For a multidimensional map nest, the same idea applies:

$$\text{segmented\_iota } [[1,2],[3,4]] = [[[0],[0,1]],[[0,1,2],[0,1,2,3]]]$$

Thus, segmented_iota preserves the outer map structure, but creates irregular arrays at the innermost level.

3.4.2 Flattening Sum

Similarly, consider a map nest whose innermost expression is a sum:

$$\text{map } (\lambda x_1 \to \cdots \text{map } (\lambda xs \to \text{sum } xs)\ xss_p \cdots)\ xss_1$$

Here, the innermost variable xs is an array. Flattening replaces the nested sums by a segmented_sum operation. The abstract meaning of segmented_sum is that it sums each innermost array, again preserving the outer structure. For example:

$$\text{segmented\_sum } [[1,2],[1,2,3]] = [3,6]$$

3.4.3 Applying the Flattening Rules

Applying the two flattening rules to the distributed running example replaces the map containing iota with segmented_iota, and the map containing sum with segmented_sum:

$$\begin{aligned}&\text{let } zss : [k][]\text{int} = \text{segmented\_iota } is\\&\text{let } ress : [k]\text{int} = \text{segmented\_sum } zss\\&\text{in } ress\end{aligned}$$

3.5 Representation of Irregular Arrays

So far, we have introduced irregular arrays only abstractly. In particular, we have not yet discussed how such arrays are represented in memory. A direct implementation of an irregular array would typically use an outer array of pointers, where each pointer refers to a separate inner array. However, this representation does not translate well to GPUs and other parallel hardware, since such architectures are most efficient when operating on contiguous, flat data.

Therefore, we represent irregular arrays using a flat data array together with segment metadata that describes how to recover the logical segments. The rest of this section first describes the basic representation for a one-dimensional irregular array, and then explains how the same representation is used for multidimensional arrays.

3.5.1 One-Dimensional Irregular Arrays

An irregular array x is represented by a tuple $\langle x_S, x_F, x_O, x_D \rangle$, where x_S is the segment-size array, x_F is the segment-flag array, x_O is the segment-offset array, and x_D is the flat data array. When the value is clear from context we drop the prefix and write $\langle S, F, O, D \rangle$.

If the irregular value has k segments, then S and O have length k. For each segment j, $S[j]$ gives the number of scalar elements in that segment, and $O[j]$ gives the position where the segment starts in the flat data array D. Thus, the element at local index q in segment j is stored at flat position $O[j] + q$ and read as $D[O[j] + q]$. The offset array is the exclusive prefix sum of the segment sizes:

$$O[j] = \sum_{q<j} S[q]$$

and the total number of flat elements is:

$$m = \sum_{j=0}^{k-1} S[j]$$

Therefore, D has length m. The flag array F also has length m, and marks the first element of each non-empty segment. The flag array is particularly useful for implementing segmented operations, such as segmented scans, on the flat data array. Figure 3.3 illustrates the representation of the irregular array $[[0], [0, 1], [0, 1, 2]]$.

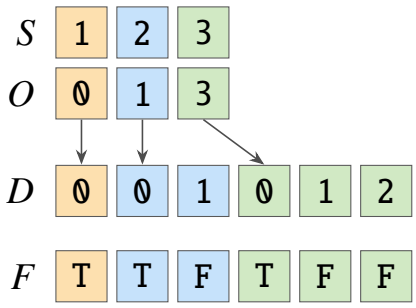


Figure 3.3: The representation of the irregular array $[\,[0],\ [0, 1],\ [0, 1, 2]\,]$ by the tuple $\langle S, F, O, D \rangle$ of segment sizes, flags, offsets, and flat data. Colours indicate which segment each element describes or belongs to, and the arrows show that $O[j]$ gives the flat position in D at which segment j starts.

It is sometimes useful to derive further structure arrays from the representation. Two common such arrays are the segment-index array and the intra-segment-index array, both of length m. The segment-index array II_1 maps each flat element to the segment it belongs to:

$$II_1[p] = j \quad \text{iff} \quad O[j] \leq p < O[j] + S[j]$$

and the intra-segment-index array II_2 maps each flat element to its local index inside that segment:

$$II_2[p] = p - O[II_1[p]]$$

For the example above:

$$II_1 = [0, 1, 1, 2, 2, 2] \qquad II_2 = [0, 0, 1, 0, 1, 2]$$

The important point is that the irregular value is represented using only flat arrays. This makes it possible to implement operations on irregular arrays using flat parallel kernels.

3.5.2 Multidimensional Irregular Arrays

The same representation can also be used when the elements of an irregular array are themselves arrays. The important point is that the segment metadata describes only the outer irregular dimension. Each segment may contain a regular multidimensional array, but the scalar elements of that array are stored contiguously in the flat data array. For example, consider the irregular array:

$$xss = [\,[[1, 2],\ [3, 4]],\ [[5, 6]],\ [[7],\ [8],\ [9]]\,]$$

The outer array has three segments. The first segment has shape 2×2, the second has shape 1×2, and the third has shape 3×1. Since the segment-size array stores the number of scalar elements in each segment, we have:

$$S = [4, 2, 3] \qquad O = [0, 4, 6]$$

$$F = [T, F, F, F, T, F, T, F, F] \qquad D = [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

No information is lost, because the inner shape of each segment is still known from the size variables that describe the produced arrays. In this example, if the source program produces arrays of type $[n][m]\mathsf{int}$, then the two inner dimensions are represented after distribution by the arrays:

$$ns = [2, 1, 3] \qquad ms = [2, 2, 1]$$

Both n and m may be variant, so in general both inner dimensions are described by such per-segment size arrays. For segment j, the logical segment has shape $ns[j] \times ms[j]$, and its scalar elements are stored in D starting at offset $O[j]$. For example, segment 0 has shape 2×2 and starts at offset 0, so it is recovered from $D[0 : 4] = [1, 2, 3, 4]$ and interpreted as $[[1, 2], [3, 4]]$.

Our representation therefore has one level of segment metadata. It does not store separate segment metadata for every inner dimension. Instead, the outer irregular structure is described by S, F, and O, while the inner regular structure of each segment is described by the corresponding size variables.

In the example above, the array inside each segment was regular. A careful reader might ask what happens if we have an irregular array inside each segment. The answer is that, since every array in the source language is regular, an individual source-language iteration cannot itself produce an irregular array. A single distributed map can therefore introduce only one irregular dimension. Multiple levels of irregularity

arise only from multiple levels of maps. In that case, the transformation does not represent the result as a nested tree of irregular arrays. Instead, the relevant map levels are collapsed into a single sequence of segments. For example, after distribution we may have an intermediate value $zss = \mathsf{segmented\_iota}\ is$, which for $is = [1, 2, 3]$ gives $zss = [[0], [0, 1], [0, 1, 2]]$. Now suppose that a later distributed statement maps over each segment of zss:

$$\mathsf{let}\ tsss = \mathsf{map}\ (\lambda zs \rightarrow \mathsf{map}\ (\lambda z \rightarrow \mathsf{iota}\ z)\ zs)\ zss$$

Conceptually, for $is = [1, 2, 3]$, this produces $[\,[[]],\ [[], [0]],\ [[], [0], [0, 1]]\,]$ However, the implementation does not represent this value as a nested tree of irregular arrays. The inner map is flattened over the flat data array of zss. Since zss has segment sizes $[1, 2, 3]$, the flattened inner map has $6 = 1 + 2 + 3$ iterations. Therefore, the result is represented as an irregular array with six segments:

$$[\,[],\ [],\ [0],\ [],\ [0],\ [0, 1]\,]$$

whose representation is:

$$S = [0, 0, 1, 0, 1, 2] \qquad O = [0, 0, 0, 1, 1, 2]$$

$$F = [T, T, T, F] \qquad D = [0, 0, 0, 1]$$

As a result, a single level of segment metadata is always sufficient.

3.6 Lowering Segmented Operations

After introducing the irregular representation, we can now show how the abstract segmented operations produced by flattening are lowered to operations on flat arrays and segment metadata. At this stage, the running example is:

$$\begin{aligned}&\mathsf{let}\ zss : [k][]\mathsf{int} = \mathsf{segmented\_iota}\ (is : [k]\mathsf{int})\\ &\mathsf{let}\ ress : [k]\mathsf{int} = \mathsf{segmented\_sum}\ (zss : [k][]\mathsf{int})\\ &\mathsf{in}\ ress\end{aligned}$$

The goal is to lower both segmented_iota and segmented_sum to flat data-parallel operations. In particular, the intermediate value zss is never materialised as a nested irregular array. Instead, it is represented by the tuple $\langle zss_S, zss_F, zss_O, zss_D \rangle$.

3.6.1 Segmented Scan as a Building Block

A central building block for the lowering is segmented scan [12]. An inclusive scan computes prefix sums

$$\mathsf{scan}\ (+)\ 0\ [1, 2, 3] = [1, 3, 6]$$

and the exclusive variant shifts the result to the right and inserts the neutral element

$$\mathsf{scan}_{exc}\ (+)\ 0\ [1, 2, 3] = [0, 1, 3]$$

We use scan here only as a helper to simplify the presentation. In the target language, these scans are expressed using segscan.

A segmented scan performs the same operation independently inside each segment. For example:

$$\text{segmented\_scan } (+)\ 0\ [[2],[3,2],[1,2,3]] = [[2],[3,5],[1,3,6]]$$

and the exclusive segmented scan gives:

$$\text{segmented\_scan}_{exc}\ (+)\ 0\ [[2],[3,2],[1,2,3]] = [[0],[0,3],[0,1,3]]$$

In our irregular representation, a segmented scan only changes the flat data array. The segment metadata, namely the segment sizes, offsets, and flags, remains unchanged. A segmented scan can be implemented using an ordinary scan and the flag array [3]. To achieve this, we define a new binary operator for the scan. For addition, we define a binary operator $\oplus$ over value–flag pairs:

$$(v_1, f_1) \oplus (v_2, f_2) = (\text{if } f_2 \text{ then } v_2 \text{ else } v_1 + v_2,\ f_1 \vee f_2)$$

The flag f_2 indicates whether the right-hand input starts a new segment. If it does, the accumulated value is reset to v_2. Otherwise, the values are combined normally. The second component records whether a segment boundary has been encountered in the scanned prefix.

Using this operator, we define a helper function segPrefixSum. It takes a flag array and a flat data array, and returns the flat data array obtained by performing a segmented prefix sum. We write scan as operating on multiple arrays, so scanning the arrays $vals$ and $flags$ with the operator $\oplus$ produces two arrays, the scanned values and the scanned flags:

$$\begin{aligned}&\text{fun segPrefixSum } [n]\ (flags : [n]\text{bool})\ (vals : [n]\text{int}) : [n]\text{int} =\\ &\quad \text{let } (res : [n]\text{int},\ flags' : [n]\text{bool}) = \text{scan } \oplus\ (0, \text{false})\ vals\ flags\\ &\quad \text{in } res\end{aligned}$$

The point is that the segmented scan can be implemented by an ordinary scan whose elements carry both a value and a segment-start flag.

The exclusive segmented scan can be obtained from the inclusive one by shifting the scanned values one position to the right within each segment and inserting the neutral element at every segment start. We write $\text{segPrefixSum}_{exc}$ for the exclusive variant.

3.6.2 Constructing the Flag Array

The helper operation gen_flags constructs the flag array from the total flat size m, the segment sizes S, and the offsets O. The flag array is defined by:

$$F[p] = \begin{cases} T, & \text{if } \exists j.\ p = O[j] \wedge S[j] > 0, \\ F, & \text{otherwise,} \end{cases}$$

so $F[p]$ is true exactly when p is the first flat position of a non-empty segment. Operationally, gen_flags can be implemented by a flat scatter. We first create an array containing only F and then, for every non-empty segment j, we write T at position $O[j]$. All other positions remain F.

3.6.3 Lowering Segmented Iota

Consider the operation zss = segmented_iota $(is : [k]\text{int})$. The input array is gives the size of each generated segment, so the segment-size array of zss is simply is.

The offsets, total size, and flags follow as usual:

$$zss_O = \text{scan}_{exc}\ (+)\ 0\ zss_S \qquad m = \text{sum}\ zss_S \qquad zss_F = \text{gen\_flags}\ m\ zss_S\ zss_O$$

The remaining component is the flat data array zss_D. The result of segmented_iota should contain the local index of each element inside its segment, which is computed by an exclusive segmented scan over an array of ones:

$$\begin{aligned}&\text{let } ones = \text{replicate } m\ 1\\ &\text{let } zss_D = \text{segPrefixSum}_{exc}\ zss_F\ ones\end{aligned}$$

For the concrete input $is = [1, 2, 3]$ we get:

$$zss_S = [1, 2, 3] \qquad zss_O = [0, 1, 3] \qquad m = 6$$

the flag array is $zss_F = [T, T, F, T, F, F]$, and the flat data array is:

$$zss_D = \text{segPrefixSum}_{exc}\ zss_F\ [1, 1, 1, 1, 1, 1] = [0, 0, 1, 0, 1, 2]$$

which is exactly the flat representation of $[[0], [0, 1], [0, 1, 2]]$.

3.6.4 Lowering Segmented Sum

Now consider the operation $ress$ = segmented_sum $(zss : [k][]\text{int})$, where zss is represented by its four flat components $\langle zss_S, zss_F, zss_O, zss_D\rangle$. The segmented sum consumes this representation directly. We first perform an inclusive segmented scan over the flat data:

$$scan\_res\_D = \text{segPrefixSum}\ zss_F\ zss_D$$

After this scan, the last element of each non-empty segment contains the sum of that segment. Therefore, the final result is obtained by a flat map over the segment sizes and offsets:

$$\begin{aligned}&red\_res\_D : [k]\text{int} = \text{segmap}\langle j < k\rangle\\ &\quad \text{let } s : \text{int} = zss_S[j]\\ &\quad \text{let } o : \text{int} = zss_O[j]\\ &\quad \text{let } isEmpty : \text{bool} = s = 0\\ &\quad \text{let } end : \text{int} = o + s\\ &\quad \text{let } idx : \text{int} = end - 1\\ &\quad \text{in if } isEmpty \text{ then } 0 \text{ else } scan\_res\_D[idx]\end{aligned}$$

If a segment is empty, the result is the neutral element 0. Otherwise, the result is read from the last flat position belonging to that segment.

For the running example, we have:

$$zss_S = [1, 2, 3] \qquad zss_O = [0, 1, 3]$$

$$zss_F = [T, T, F, T, F, F] \qquad zss_D = [0, 0, 1, 0, 1, 2]$$

the segmented scan gives:

$$scan\_res\_D = \mathsf{segPrefixSum}\ zss_F\ zss_D = [0, 0, 1, 0, 1, 3]$$

and taking the last value of each segment gives $red\_res\_D = [0, 1, 3]$.

Since the type of $ress$ is regular, the result can be represented by a single data array. The array red_res_D has exactly the required type $[k]\mathsf{int}$. Therefore, the result of the original program is:

$$ress = red\_res\_D = [0, 1, 3]$$

3.6.5 The Flat Program

Putting the two lowerings together, the running example is transformed into the following flat program:

$\mathsf{let}\ zss_S : [k]\mathsf{int} = is$
$\mathsf{let}\ zss_O : [k]\mathsf{int} = \mathsf{scan}_{exc}\ (+)\ 0\ zss_S$
$\mathsf{let}\ m : \mathsf{int} = \mathsf{sum}\ zss_S$
$\mathsf{let}\ zss_F : [m]\mathsf{bool} = \mathsf{gen\_flags}\ m\ zss_S\ zss_O$
$\mathsf{let}\ ones : [m]\mathsf{int} = \mathsf{replicate}\ m\ 1$
$\mathsf{let}\ zss_D : [m]\mathsf{int} = \mathsf{segPrefixSum}_{exc}\ zss_F\ ones$
$\mathsf{let}\ scan\_res\_D : [m]\mathsf{int} = \mathsf{segPrefixSum}\ zss_F\ zss_D$
$\mathsf{let}\ red\_res\_D : [k]\mathsf{int} = \mathsf{segmap}\langle j < k\rangle$
    $\mathsf{let}\ s = zss_S[j]$
    $\mathsf{let}\ o = zss_O[j]$
    $\mathsf{in\ if}\ s = 0\ \mathsf{then}\ 0\ \mathsf{else}\ scan\_res\_D[o + s - 1]$
$\mathsf{in}\ red\_res\_D$

At this point, the program contains only regular arrays, every parallel statement is a flat data-parallel operation, and there is no remaining nested parallelism. Thus, the transformation has achieved its goal.

3.7 Nested Maps and Free Variables

The running example so far had no free variables inside its maps and every variable used by an inner operation was bound within the same map iteration. In general,

however, an inner map may refer to a variable bound by an outer map. We now extend the running example to show how flattening handles such free variables, using the following program:

$$
\begin{array}{l}
e : [k]\mathsf{int} = \\
\quad \mathsf{map}\ (\lambda(i : \mathsf{int}) \rightarrow \\
\quad\quad \mathsf{let}\ xs : [i]\mathsf{int} = \mathsf{iota}\ i \\
\quad\quad \mathsf{let}\ ys : [i]\mathsf{int} = \mathsf{map}\ (\lambda(x : \mathsf{int}) \rightarrow x + i)\ xs \\
\quad\quad \mathsf{let}\ r : \mathsf{int} = \mathsf{sum}\ ys \\
\quad\quad \mathsf{in}\ r)\ is
\end{array}
$$

3.7.1 Distribution

The first and third statements of the outer map are the same as in the running example of Section 3.2, and they are distributed in the same way. The second statement contains the inner map. Since the inner map uses both xs and the outer value i, the distributed map receives both is and xss. Thus, after distribution, the program has the form:

$$
\begin{array}{l}
\mathsf{let}\ xss : [k][]\mathsf{int} = \mathsf{map}\ (\lambda(i : \mathsf{int}) \rightarrow \mathsf{iota}\ i)\ is \\
\mathsf{let}\ yss : [k][]\mathsf{int} = \\
\quad \mathsf{map}\ (\lambda(i : \mathsf{int})\ (xs : []\mathsf{int}) \rightarrow \\
\quad\quad \mathsf{map}\ (\lambda(x : \mathsf{int}) \rightarrow x + i)\ xs)\ is\ xss \\
\mathsf{let}\ ress : [k]\mathsf{int} = \mathsf{map}\ (\lambda(ys : []\mathsf{int}) \rightarrow \mathsf{sum}\ ys)\ yss \\
\mathsf{in}\ ress
\end{array}
$$

3.7.2 Flattening the Inner Map

The first thing to notice is that the number of iterations of the inner map may vary between iterations of the outer map. In this example, the inner map has i iterations, and i is variant because it changes between iterations of the outer map. We call the inner map non-uniform because its width is variant with respect to the surrounding map nest.

After distribution, the different values of i across the outer map are represented by the array is. Thus, is is the per-iteration representation of the outer variable i, and in this example it also gives the number of iterations of the inner maps. For $is = [1, 2, 3]$, the first outer iteration has one inner-map iteration, the second has two, and the third has three.

The goal is to replace the nested inner maps by one flat map. The total number of iterations of this flat map is $m = \mathsf{sum}\ is$. For $is = [1, 2, 3]$ this gives $m = 6$. The elements that the inner maps iterate over are stored in the flat data array xss_D, so the inner map becomes a flat map over xss_D. In this example, the inner-map widths are the same as xss_S, because xs is a one-dimensional array and the inner map is applied directly to it.

However, there is still a problem. The body of the inner map uses i, in the function $\lambda(x : \mathsf{int}) \rightarrow x + i$. The value i is available once per outer iteration, but after flattening

we need one value of i for each flat inner-map iteration. Therefore, we must replicate the per-iteration representation of i, namely is, according to the number of iterations of the inner maps. We define another segmented operation, segmented_rep:

$$\text{segmented\_rep}\ [s_0, \ldots, s_{k-1}]\ [v_0, \ldots, v_{k-1}] = [\underbrace{v_0, \ldots, v_0}_{s_0 \text{ times}}, \ldots, \underbrace{v_{k-1}, \ldots, v_{k-1}}_{s_{k-1} \text{ times}}]$$

For example, segmented_rep $[1, 2, 3]$ $[1, 2, 3] = [1, 2, 2, 3, 3, 3]$

Lowering Segmented Replication

To lower segmented_rep, we use the segment-index array II_1. In this case, the segmentation is given by the replication sizes. For the replication sizes $[1, 2, 3]$, the corresponding segment-index array is $II_1 = [0, 1, 1, 2, 2, 2]$.

This array tells us which original value should be used at each position of the replicated result. Therefore, segmented_rep can be lowered to a map that indexes into the original value array:

$$\text{segmented\_rep}\ [1, 2, 3]\ is = \text{segmap}\langle p < m \rangle\ is[II_1[p]]$$

For $is = [1, 2, 3]$, this gives $[is[0], is[1], is[1], is[2], is[2], is[2]] = [1, 2, 2, 3, 3, 3]$

The remaining question is how to construct II_1. We first compute the offsets and the total flat size from the replication sizes:

$$O = \text{scan}_{exc}\ (+)\ 0\ [1, 2, 3] = [0, 1, 3] \qquad m = 6$$

Then we create an array of length m that stores the segment number at the beginning of each non-empty segment:

$$starts = [0, 1, 0, 2, 0, 0]$$

and finally a segmented prefix sum propagates each segment number until the next segment start:

$$II_1 = \text{segPrefixSum}\ flags\ starts = [0, 1, 1, 2, 2, 2]$$

This is the same segment-index array introduced in Section 3.5, but here it is constructed for the replication sizes. We write gen_segment_indices for this construction, so the lowering of the replication in the running example is:

$$\begin{aligned} &\text{let } II_1 : [m]\text{int} = \text{gen\_segment\_indices}\ is \\ &\text{let } is_{rep} : [m]\text{int} = \text{segmap}\langle p < m \rangle\ is[II_1[p]] \end{aligned}$$

The Flattened Inner Map

Now the flattened inner map can use the replicated array:

$$\text{let } yss_D : [m]\text{int} = \text{segmap}\langle p < m \rangle\ xss_D[p] + is_{rep}[p]$$

In this example, the inner map produces one scalar result for each element of xs, so the segment metadata of yss is the same as that of xss, and only the flat data array changes:

$$yss \equiv \langle xss_S,\ xss_F,\ xss_O,\ yss_D \rangle$$

For the concrete input, $xss_D = [0, 0, 1, 0, 1, 2]$ and $is_{rep} = [1, 2, 2, 3, 3, 3]$. Therefore, we have:

$$yss_D = \mathsf{segmap}\langle p < 6 \rangle\ [0, 0, 1, 0, 1, 2][p] + [1, 2, 2, 3, 3, 3][p] = [1, 2, 3, 3, 4, 5]$$

which is the flat data representation of $[[1], [2, 3], [3, 4, 5]]$.

Before lowering the segmented operations, the flattened program is:

$$\begin{aligned}
&\mathsf{let}\ xss : [k][]\mathsf{int} = \mathsf{segmented_iota}\ is \\
&\mathsf{let}\ is_{rep} : [m]\mathsf{int} = \mathsf{segmented_rep}\ is\ is \\
&\mathsf{let}\ yss_D : [m]\mathsf{int} = \mathsf{segmap}\langle p < m \rangle\ xss_D[p] + is_{rep}[p] \\
&\mathsf{let}\ yss : [k][]\mathsf{int} = \langle xss_S,\ xss_F,\ xss_O,\ yss_D \rangle \\
&\mathsf{let}\ ress : [k]\mathsf{int} = \mathsf{segmented_sum}\ yss \\
&\mathsf{in}\ ress
\end{aligned}$$

3.7.3 The Replication Problem

This example illustrates one of the central costs of flattening. The issue arises when an inner map uses a free variable from an outer map nest. If the free variable is invariant with respect to the outer map nest, then the flattened inner operation can use it directly. However, if the free variable is variant, then each inner iteration must be associated with the value from the corresponding outer iteration. A value that was computed once per outer iteration must therefore be made available at every corresponding inner iteration. In the example, this was done by constructing is_{rep}. We call this the *replication problem*.

The cost has two parts. The first is the execution overhead of segmented replication itself. The second, and more serious, is memory traffic. The replicated value must be materialised in full, which can be much larger than the original value.

The memory penalty becomes more severe as the nesting depth increases. Each level of the map nest at which an outer value is used forces another replication across the flat iteration space of that level, and the flat iteration space grows with every level. Consider a three-deep nest in which an outermost value is used at the innermost level. The value must first be replicated across the iterations of the middle level, and the result must then be replicated again across the iterations of the innermost level, so it may be far larger than the original value. An outermost value of constant size can be expanded into an array whose size is proportional to the total work of the entire nest.

This replicated data can become a dominant source of memory traffic in a flattened program, even though it carries no information beyond the original outer value. Avoiding such unnecessary replication is therefore one of the costs addressed by this thesis.

Chapter 4

Avoiding the Costs of Flattening

The transformation of Chapter 3 flattens all nested parallelism in a program, but it does so at a cost. Full flattening materialises intermediate arrays, replicates values across flattened iteration spaces, and dissolves regular structure into segment metadata, which can hurt locality and increase memory traffic [16, 20]. Full flattening also exploits all available parallelism even when the hardware cannot use it [5]. This chapter shows how these costs can sometimes be avoided.

The flattening of an inner map replicates each variant free variable across the flattened iteration space, as described in Section 3.7.3. We begin by showing how that replication can often be left implicit rather than materialised (Section 4.1). We then observe that purely scalar computation need not be distributed at all (Section 4.2), and that when the inner parallelism is uniform, the nest should be kept as a regular multidimensional operation rather than dissolved into segment metadata (Section 4.3). We also show that uniform sums can be lowered directly to regular segmented reductions (Section 4.4). Finally, we discuss incremental flattening, which avoids exploiting more parallelism than the hardware can use by generating several versions of the code and choosing between them at run time (Section 4.5).

4.1 Replication Avoidance

The previous chapter showed the replication problem. A variable that is available once per outer iteration may have to be replicated to the flattened iteration space of an inner map. In the example, the variable i is bound by the outer map but used inside the inner map, so after flattening we needed one value of i for each flat iteration of the inner map.

The direct solution is to materialise this replication. As shown in Section 3.7, this can be done by constructing the segment-index array II_1 and then indexing into the original array:

$$\begin{aligned}&\mathsf{let}\ II_1 : [m]\mathsf{int} = \mathsf{gen\_segment\_indices}\ is\\ &\mathsf{let}\ is_{rep} : [m]\mathsf{int} = \mathsf{segmap}\langle p < m\rangle\ is[II_1[p]]\end{aligned}$$

For $is = [1, 2, 3]$, this gives $is_{rep} = [1, 2, 2, 3, 3, 3]$. The flattened inner map can then

use is_{rep} as an ordinary flat array. However, this materialises an intermediate array whose only purpose is to provide the correct value of i to each flat iteration. As mentioned, this can lead to space overhead, especially when the replicated value is large or when this happens at several levels of nesting.

4.1.1 Avoiding Replication of Regular Values

Instead of materialising is_{rep}, we can keep the original array is and use II_1 directly in the map that consumes the replicated value. At flat position p, the value of i that would have been stored in $is_{rep}[p]$ is instead read as $is[II_1[p]]$.

Thus, instead of first constructing is_{rep} and then using it in the flattened inner map, we can write:

$$\begin{aligned}&\textsf{let } II_1 : [m]\textsf{int} = \textsf{gen\_segment\_indices } is\\ &\textsf{let } yss_D : [m]\textsf{int} = \textsf{segmap}\langle p < m\rangle\\ &\quad\textsf{let } i = is[II_1[p]]\\ &\quad\textsf{in } xss_D[p] + i\end{aligned}$$

This has the same effect as using is_{rep}, but the replicated array is not materialised. The indexed read that would have been used to construct the replicated array is fused into the flat map that consumes the replicated value.

4.1.2 Avoiding Replication of Irregular Values

The same idea can be used when the value that must be replicated is itself an irregular array. In this case, materialising the replication can be even more expensive, because it may require copying not only scalar values, but entire segments of a flat data array.

Consider the irregular array:

$$[\,[5],\ [6,7,9],\ [10,14]\,]$$

Its flat representation is:

$$S = [1,3,2] \qquad O = [0,1,4]$$

$$F = [T,T,F,F,T,F] \qquad D = [5,6,7,9,10,14]$$

Now assume that this irregular array must be replicated according to map width, $W = [8,3,3]$. If we materialised the replication, the logical result would contain eight copies of the first segment, three of the second, and three of the third:

$$\begin{aligned}&[\,[5],[5],[5],[5],[5],[5],[5],[5],\\ &\ [6,7,9],[6,7,9],[6,7,9],\\ &\ [10,14],[10,14],[10,14]\,].\end{aligned}$$

The materialised representation would have segment sizes:

$$S' = [1,1,1,1,1,1,1,1,3,3,3,2,2,2]$$

and would require a replicated data array:

$$D' = [5, 5, 5, 5, 5, 5, 5, 5, 6, 7, 9, 6, 7, 9, 6, 7, 9, 10, 14, 10, 14, 10, 14]$$

This is expensive because the flat data array has been copied.

Instead, we can avoid materialising D'. The key observation is that every replicated segment is still a copy of a segment that already exists in the original data array D. Therefore, instead of copying the data, we can keep D unchanged and replicate only the metadata needed to read from it.

First, we replicate the segment sizes:

$$S_{rep} = \mathsf{segmented\_rep}\ W\ S = [1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 2, 2, 2]$$

which is the same segment-size array that the fully materialised replicated irregular array would have. Next, we replicate the original offsets:

$$O_{rep} = \mathsf{segmented\_rep}\ W\ O = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 4, 4, 4]$$

The array O_{rep} is not the usual offset array of an irregular array. In a standard representation, the offset array is the exclusive prefix sum of the segment sizes. Here, O_{rep} instead points back into the original data array D. The first eight replicated segments all use offset 0, because they are copies of the original segment $[5]$. The next three use offset 1, and the last three use offset 4.

Finally, the flag array requires no replication at all. Since the data array D is shared, the original flag array still describes it, and we keep it unchanged:

$$F_{rep} = F = [T, T, F, F, T, F]$$

Thus, the replicated irregular value can be represented without copying the flat data array, as (S_{rep}, F, O_{rep}, D).

To read element q of replicated segment j, we use the source offset:

$$D[O_{rep}[j] + q]$$

which gives the same value as reading from the fully materialised replicated array, but without constructing D'.

This representation is useful when the replicated irregular value is only read. A flattened map that consumes it can read directly from the original data array using O_{rep}, so the expensive part of the replication, namely copying the flat data array, is avoided. We may still need to construct replicated metadata such as S_{rep} and O_{rep}, but this can be much smaller than the full replicated data array, especially when the segments are large.

If the program needs a standard irregular representation, for example because the value is updated or passed to an operation that requires ordinary offsets, then the replication may have to be materialised.

4.2 Vectorisation Avoidance

In the description of distribution, we said that the goal was to transform a map body into a sequence of maps, each containing a single statement. This is a useful simplification for explaining the basic flattening transformation, but it is not always a good implementation strategy. In particular, distributing every statement can create unnecessary intermediate arrays. An analogous optimisation is performed in Data Parallel Haskell [20]. The main idea is that not every statement in a map body needs to be vectorised. Some statements are scalar computations that can simply remain inside the surrounding map.

Consider the following simple example:

$$
\begin{array}{l}
\mathsf{map}\ (\lambda x \rightarrow \\
\quad \mathsf{let}\ i = x + 100 \\
\quad \mathsf{let}\ t = i * 7 \\
\quad \mathsf{in}\ t)\ xs
\end{array}
$$

If distribution is applied blindly, this program is transformed into two separate maps:

$$
\begin{array}{l}
\mathsf{let}\ is = \mathsf{map}\ (\lambda x \rightarrow x + 100)\ xs \\
\mathsf{let}\ ts = \mathsf{map}\ (\lambda i \rightarrow i * 7)\ is \\
\mathsf{in}\ ts
\end{array}
$$

This introduces the intermediate array is. However, the computation $x + 100$ is just a scalar operation. It does not introduce nested parallelism, and it does not construct an irregular array. Therefore, there is no need to split it into a separate map. It is better to leave the two scalar statements together inside the same map,

$$
\begin{array}{l}
\mathsf{map}\ (\lambda x \rightarrow \\
\quad \mathsf{let}\ i = x + 100 \\
\quad \mathsf{let}\ t = i * 7 \\
\quad \mathsf{in}\ t)\ xs.
\end{array}
$$

During distribution, we therefore distinguish between scalar statements and parallel statements. We use the term parallel statement for any statement that either contains a parallel operation or produces a value that may require flattening. This includes operations such as map, iota, sum and rearrange, as well as statements that contain such operations inside their body. For example, a conditional whose branch contains a map cannot be treated as a scalar statement. We also treat statements that produce values whose type contains a variant size as parallel statements, since such values become irregular after distribution.

All other statements are classified as scalar. These are statements whose expressions perform only scalar computation, such as arithmetic or boolean operations, and whose result does not introduce a variant-size array. Consecutive scalar statements are grouped together and kept inside the same map body, so that instead of creating one map for each scalar statement, the compiler emits one map for the whole scalar group.

This avoids creating intermediate arrays for values that are only used inside the same map iteration. It also avoids launching extra kernels for computations that can be performed inside an existing kernel. This makes vectorisation avoidance an important optimisation in practice, since scalar computations often occur between the parallel operations that actually need to be flattened.

4.3 Uniform Nested Map

Section 3.7 showed how a non-uniform inner map is flattened. In that example, the amount of inner parallelism could vary between iterations of an outer map, so the flattened program needed segment metadata to describe the irregular iteration space. We now consider the uniform case, in which the inner map has the same size in every outer iteration.

This case is important because it should not be treated in the same way as the irregular case. If the nested iteration space is regular, then constructing segment sizes, offsets, and flags adds unnecessary overhead and obscures useful structure in the program. Instead, the compiler should preserve the regular multidimensional structure.

4.3.1 Running Example

A standard example is matrix multiplication. We assume that the second input matrix has already been transposed, so that the inner map iterates over columns represented as rows:

$$
\begin{aligned}
&\mathsf{fun\ matmul}\ (A : [n][m]\mathsf{int})\ (B^T : [p][m]\mathsf{int}) : [n][p]\mathsf{int} = \\
&\quad \mathsf{map}\ (\lambda(A_{row} : [m]\mathsf{int}) \rightarrow \\
&\qquad \mathsf{map}\ (\lambda(B_{col} : [m]\mathsf{int}) \rightarrow \\
&\qquad\quad \mathsf{fold}\ (\lambda(acc : \mathsf{int})\ (a : \mathsf{int})\ (b : \mathsf{int}) \rightarrow \\
&\qquad\qquad acc + a * b)\ 0\ A_{row}\ B_{col})\ B^T)\ A
\end{aligned}
$$

4.3.2 Distribution

As before, we start by distributing the map body. The body of the outer map contains only one statement, namely the inner map over B^T, so there is no map fission to perform at the outer level. The body of the inner map contains only a sequential fold, and since the fold does not introduce additional parallelism, map fission does not split it further. The result of distribution is therefore essentially the same nested map structure:

$$
\begin{aligned}
&\textsf{let } C : [n][p]\textsf{int} = \textsf{map } (\lambda(A_{row} : [m]\textsf{int}) \rightarrow \\
&\quad \textsf{map } (\lambda(B_{col} : [m]\textsf{int}) \rightarrow \\
&\quad\quad \textsf{let } dot : \textsf{int} = \textsf{fold } (\lambda(acc : \textsf{int})\ (a : \textsf{int})\ (b : \textsf{int}) \rightarrow \\
&\quad\quad\quad\quad acc + a * b)\ 0\ A_{row}\ B_{col} \\
&\quad\quad \textsf{in } dot)\ B^T)\ A \\
&\textsf{in } C
\end{aligned}
$$

4.3.3 Flattening the Inner Map

We now flatten the inner map. The inner map has width p, because it maps over $B^T : [p][m]\textsf{int}$. Unlike the non-uniform example, the width p is invariant with respect to the outer map, and each of the n outer iterations contains an inner map of the same width p. Therefore, the nested map has a regular iteration space of shape $n \times p$.

This is the key difference from the irregular case. Since the inner width is invariant, there is no need for a per-iteration width array such as $[1, 2, 3]$ for a non-uniform map, and therefore no need to construct segment sizes, offsets, or flags. Instead, we can represent the nested map by a multidimensional segmented map in the target language, written $\textsf{segmap}\langle i < n, j < p\rangle$. This denotes a regular multidimensional map space with two logical indices i and j. Although it is written with two logical indices, it is still a flat kernel map, and the indices simply describe the regular iteration space of that kernel.

We still have to deal with the free variables used by the inner map. In particular, the inner map uses A_{row}, which is bound by the outer map. After flattening, this row must be available for every inner index j. Conceptually, this means that A is lifted to the new regular map space, so that $A_{rowrep}[i, j] = A[i]$. Instead of using segmented replication and segment metadata, this lifting can be expressed with ordinary replication and rearrangement:

$$
\begin{aligned}
&\textsf{let } A_{rep} : [p][n][m]\textsf{int} = \textsf{replicate } p\ A \\
&\textsf{let } A_{rowrep} : [n][p][m]\textsf{int} = \textsf{rearrange}\langle 1, 0, 2\rangle\ A_{rep}
\end{aligned}
$$

The replication creates one copy of A for each inner-map iteration, giving an array of shape $[p][n][m]\textsf{int}$, and the rearrangement moves the replicated dimension after the original outer dimension, giving the desired shape $[n][p][m]\textsf{int}$.

We also have to lift the input of the inner map. The inner map iterates over B^T, which is independent of the outer index i, so the same B^T must be available for every row of A. Conceptually, $B^T_{rep}[i, j] = B^T[j]$, which is obtained by replicating B^T with the outer-map width:

$$
\textsf{let } B^T_{rep} : [n][p][m]\textsf{int} = \textsf{replicate } n\ B^T
$$

Using these lifted inputs, the flattened target program is:

$$
\begin{aligned}
&\textsf{let } A_{rep} : [p][n][m]\textsf{int} = \textsf{replicate } p\ A \\
&\textsf{let } A_{rowrep} : [n][p][m]\textsf{int} = \textsf{rearrange}\langle 1,0,2\rangle\ A_{rep} \\
&\textsf{let } B^T_{rep} : [n][p][m]\textsf{int} = \textsf{replicate } n\ B^T \\
&\textsf{let } C : [n][p]\textsf{int} = \textsf{segmap}\langle i < n, j < p\rangle \\
&\qquad \textsf{let } A_{row} : [m]\textsf{int} = A_{rowrep}[i,j] \\
&\qquad \textsf{let } B_{col} : [m]\textsf{int} = B^T_{rep}[i,j] \\
&\qquad \textsf{let } dot : \textsf{int} = \textsf{fold } (\lambda(acc : \textsf{int})\ (a : \textsf{int})\ (b : \textsf{int}) \rightarrow \\
&\qquad\qquad acc + a * b)\ 0\ A_{row}\ B_{col} \\
&\qquad \textsf{in } dot \\
&\textsf{in } C
\end{aligned}
$$

This program is flat with respect to the two outer levels of parallelism. The source-level inner map has disappeared, and its parallelism is represented by the second dimension of the target segmap.

4.3.4 Simplification

The replications used to construct A_{rowrep} and B^T_{rep} are administrative. They make the lifted inputs match the regular map space of the flattened program, but they do not necessarily lead to materialised copies.

Since B^T_{rep} is obtained by replicating B^T along the outer dimension, the access $B^T_{rep}[i,j]$ simplifies to $B^T[j]$. Similarly, since A_{rowrep} is obtained by replicating A and then rearranging the replicated dimension, the access $A_{rowrep}[i,j]$ simplifies to $A[i]$. Therefore a simple simplifier can transform the program to the following program:

$$
\begin{aligned}
&\textsf{let } C : [n][p]\textsf{int} = \textsf{segmap}\langle i < n, j < p\rangle \\
&\qquad \textsf{let } A_{row} : [m]\textsf{int} = A[i] \\
&\qquad \textsf{let } B_{col} : [m]\textsf{int} = B^T[j] \\
&\qquad \textsf{let } dot : \textsf{int} = \textsf{fold } (\lambda(acc : \textsf{int})\ (a : \textsf{int})\ (b : \textsf{int}) \rightarrow \\
&\qquad\qquad acc + a * b)\ 0\ A_{row}\ B_{col} \\
&\qquad \textsf{in } dot \\
&\textsf{in } C
\end{aligned}
$$

We note that since the lifted inputs are constructed using regular replication and rearrangement, a simplifier can remove the administrative lifting and recover direct accesses to the original arrays.

4.3.5 Preserving Structure for GPU Memory Optimisation

The main benefit of using a regular multidimensional segmap is that it keeps the structure of the computation visible after flattening. This matters especially on GPUs, where performance is often limited by memory access rather than arithmetic. As mentioned in Section 2.1.2, unlike CPUs, GPUs do not rely on a large cache hierarchy

to hide irregular memory accesses. Instead, efficient GPU code must expose regular access patterns that can later be mapped to coalesced global-memory accesses or to explicit use of local memory.

In the matrix multiplication example, the two logical indices of $\mathsf{segmap}\langle i < n, j < p\rangle$ correspond directly to the two dimensions of the output matrix. The index i selects a row of A, and the index j selects a row of B^T. Keeping these indices explicit makes the memory access pattern of the computation visible in the target program.

This is important for later transformations such as block tiling. A tile can be formed as a rectangular block of the (i, j) iteration space. Within such a block, several output elements reuse the same rows of A and B^T, so the compiler can load these values into faster memory and reuse them across multiple threads. This kind of transformation relies on the computation still having an analysable regular structure.

If we used the irregular segmented representation for this uniform case, the same computation would be described through segment sizes, offsets, flags, and flat data arrays. That representation is necessary when the inner widths vary, but here it would only obscure the regular matrix structure. Since the inner width is invariant, we can flatten the two maps into a regular product space and preserve the structure that later optimisations need.

4.4 Uniform Sum

In Section 3.4.2 we handled a sum inside a map by turning it into a segmented_sum, and then lowered that segmented_sum using segmented scans over the flat representation. That lowering was needed because the sum was non-uniform as the summed arrays had varying lengths.

When the sum is uniform, the segmented_sum can instead be lowered to the target construct segred, exactly as a uniform map is lowered to segmap. Consider the following expression:

$$\mathsf{map}\ (\lambda(xs : [m]\mathsf{int}) \to \mathsf{sum}\ xs)\ (xss : [n][m]\mathsf{int})$$

Here the length of xs is m in every iteration, so xss is a regular $[n][m]$int array and the sum is uniform. Rather than build segment metadata, we reduce the inner dimension of the regular array directly:

$$\mathsf{segmented\_sum}\ xss = \mathsf{segred}\langle i < n,\ j < m\rangle\ (+)\ 0\ xss[i, j]$$

This keeps the outer dimension $i < n$ and reduces along $j < m$, giving a result of type $[n]$int, one sum per row.

This is much more efficient than the non-uniform lowering. There is no flag array, no segmented scan over a flat data array, and no segment metadata, just a single regular reduction over the $n \times m$ index space. As with uniform maps, keeping the regular structure also preserves the analysable access pattern that later GPU optimisations rely on.

4.5 Incremental Flattening

We end this chapter by relating our transformation to incremental flattening [5]. Full flattening, as developed so far, exploits all of the parallelism in a program. On an idealised machine with unlimited parallel resources this is exactly what we want, since the machine can never be saturated. On real hardware such as a GPU, however, the amount of available parallelism is limited, as discussed in Chapter 2. Exploiting more parallelism than the hardware can use does not help, and it carries a cost. The flattened program materialises intermediate arrays and performs extra memory traffic that a less aggressively parallelised version would avoid. It is therefore not always a good idea to exploit all available parallelism, and full flattening on its own can perform poorly.

The difficulty is that we usually cannot tell at compile time how much parallelism is enough, because this depends on run-time values such as the width of a map. Following the approach of Henriksen et al. [5], we therefore generate several semantically equivalent versions of the code and choose between them at run time, based on the actual sizes.

4.5.1 Running Example

We use the following program as a running example:

$$\mathsf{map}\ (\lambda(xs : [m]\mathsf{int}) \to \mathsf{sum}\ xs)\ (xss : [n][m]\mathsf{int})$$

The outer map has width n, and the inner sum is a uniform sum, so the nest is in the uniform case. For this expression we generate three versions.

4.5.2 Version 1: Full Flattening

The first version is the fully flattened program developed earlier in this thesis. The uniform sum is lowered to a segred over the regular $n \times m$ index space, and exploits the full $n \times m$ parallelism:

$$\mathsf{segred}\langle i < n,\ j < m\rangle\ (+)\ 0\ xss[i, j]$$

4.5.3 Version 2: Outer Parallelism Only

Suppose the outer width n is already large enough to saturate the hardware. Then there is no benefit in also exploiting the inner parallelism, and turning the sum into a segred only adds overhead. In this version we instead parallelise only the outer map and run each inner sum sequentially.

We express this by turning the outer map into a segmap, so that the outer level runs in parallel, while leaving the sum intact inside the body. Since there is no parallel construct inside a segmap body, the sum runs sequentially:

$$\mathsf{segmap}\langle i < n\rangle\ \mathsf{sum}\ xss[i]$$

Sequentialising the sum in this way is equivalent to replacing it by the sequential fold of our source language:

$$\mathsf{segmap}\langle i < n\rangle\ \mathsf{fold}\ (\lambda(acc : \mathsf{int})\ (x : \mathsf{int}) \to acc + x)\ 0\ xss[i]$$

This version exploits only the outer parallelism of width n.

4.5.4 Version 3: Intra-Block

The third version exploits a feature of GPU hardware mentioned in Section 2.1.2. The threads of a single block can communicate cheaply through shared memory. The idea is to assign each outer map nest iteration to one block, and to let the threads of that block handle the inner parallelism for that iteration.

Assuming that the intermediate arrays needed for one outer iteration fit in shared memory, the whole sequence of statements in the map nest can execute inside a single kernel, with barriers instead of kernel launches between them. The data for that outer iteration can then be loaded from global memory once, stored in shared memory, and reused by the threads of the block. This can lead to very efficient code.

We express this version by annotating the constructs with the level at which they execute. The outer map becomes a segmap at block level, assigning one block to each segment, and the inner sum becomes a segred executed within a block,

$$\mathsf{segmap}_{block}\langle i < n\rangle\ \mathsf{segred}_{in\text{-}block}\langle j < m\rangle\ (+)\ 0\ xss[i, j]$$

The intra-block version directly maps two levels of parallelism to the GPU hierarchy. The outer level is mapped to blocks, and the inner level is mapped to threads within a block. If the map body contains deeper nested parallelism, that parallelism must be fully flattened before it can be executed within the block. This version is only possible when the memory needed for a segment fits in shared memory.

4.5.5 Generating Multi-Version Code

Maps are the expressions that give rise to multi-version code, since they are the source of the parallelism whose exploitation we wish to control. Therefore, whenever we encounter a map, we consider generating multi-version code. However, not every map can be versioned. Versioning requires uniform parallelism, and some constructs rule out particular versions. More precise conditions are tied to the implementation and are described in Section 6.8.

The choice between versions is made at run time by comparing the relevant sizes against thresholds. These thresholds are autotuned, which is important for performance. Writing w for the outer width, $m_{\min}$ for the minimum available inner parallelism, $m_{\max}$ for the maximum required thread-block size, and B for the maximum

block size, the choice can be summarised as:

$$
\begin{aligned}
&\textsf{let } \mathit{suff\_outer} = w > \mathit{threshold}_{\mathit{outer}} \\
&\textsf{let } \mathit{suff\_intra} = m_{\min} > \mathit{threshold}_{\mathit{intra}} \\
&\textsf{let } \mathit{fits} = m_{\max} \leq B \\
&\textsf{if } \mathit{suff\_outer} \\
&\quad \textsf{then } e_{\mathit{outer}} \\
&\textsf{else if } \mathit{suff\_intra} \wedge \mathit{fits} \\
&\quad \textsf{then } e_{\mathit{intra}} \\
&\textsf{else } e_{\mathit{full}}
\end{aligned}
$$

If the outer width is large enough on its own, we sequentialise the inner body and use the outer-only version e_{outer}. Otherwise, if there is enough work for the intra-block version and the inner parallelism fits within a block, we use the intra-block version e_{intra}. Finally, the fully flattened version e_{full} serves as the fall-back when neither of the other two applies.

We emphasise that incremental flattening is essential for achieving high performance. Full flattening can introduce substantial overhead, and in many cases the outer-only version avoids this overhead entirely. Depending on the specific program and input sizes, the outer-only version can therefore be much faster, sometimes by orders of magnitude, than the fully flattened version.

Chapter 5

Flattening Further Language Constructs

This chapter extends the flattening transformation to language constructs that were not covered by the core transformation. The previous chapters introduced the basic transformation (Chapter 3) and then showed how some of its costs can be avoided (Chapter 4). In this chapter, we show how to flatten three further constructs, namely conditional expressions (Section 5.1), rearrange expressions (Section 5.2), and function calls (Section 5.3). A common theme throughout the chapter is that uniform expressions can be flattened much more efficiently than non-uniform expressions. It is therefore important to identify the uniform cases.

5.1 Flattening Conditional Expressions

A conditional expression is only interesting for flattening when at least one of its branches contains parallelism. If both branches contain only scalar computation, the conditional can simply remain inside the surrounding map body.

There are two cases, depending on whether the condition is invariant or variant with respect to the surrounding map nest. If the condition is invariant, all iterations of the map nest take the same branch, and the conditional can be interchanged with the surrounding map nest. We call this a uniform conditional. If the condition is variant, different iterations may take different branches, and a different treatment is needed. We call these non-uniform conditionals.

In a distributed statement, it is easy to determine whether the condition is variant. If the condition depends on one of the inputs to the distributed conditional, then it is variant, otherwise it is invariant.

5.1.1 Uniform Conditionals

First consider the case where the condition is invariant. In the following example, b is computed outside the map and is therefore the same for every iteration:

```
let b : bool = t > 100
let res : [n][m]int = map (λ(xs : [m]int) →
      if b
          then map (λ(x : int) → x * 5) xs
          else map (λ(x : int) → x * 20) xs
    ) xss
in res
```

Here $xss : [n][m]\text{int}$. The outer map iterates over the rows of xss, while each branch of the conditional contains an inner map over a row xs. Since b is invariant, every outer-map iteration chooses the same branch. Therefore, we can exchange the conditional with the outer map:

```
let b : bool = t > 100
let res : [n][m]int =
    if b
        then
          map (λ(xs : [m]int) →
            map (λ(x : int) → x * 5) xs) xss
        else
          map (λ(xs : [m]int) →
            map (λ(x : int) → x * 20) xs) xss
in res
```

This transformation is semantics-preserving because the value of b is the same for every iteration. If b is true, then all iterations execute the then-branch, and if b is false, then all iterations execute the else-branch. Therefore, choosing the branch once outside the map gives the same result as choosing the same branch inside every map iteration.

After this exchange, each branch contains a uniform nested map. The inner map has width m in every outer iteration, so each branch can be flattened into a regular two-dimensional map space, exactly as in Section 4.3:

```
let b : bool = t > 100
let res : [n][m]int =
    if b
        then segmap⟨i < n, j < m⟩ xss[i, j] * 5
        else segmap⟨i < n, j < m⟩ xss[i, j] * 20
in res
```

5.1.2 Non-Uniform Conditionals

The previous transformation only works when the condition is invariant with respect to the surrounding map nest. We now consider the case where the condition is variant, so that different iterations of the map may take different branches. The conditional

then cannot be exchanged with the outer map, because there is no single branch taken by all iterations.

Consider the following example:

$$
\begin{aligned}
&\mathsf{let}\ res : [n][m]\mathsf{int} = \mathsf{map}\ (\lambda(xs : [m]\mathsf{int})\ (b : \mathsf{bool}) \to \\
&\quad \mathsf{if}\ b \\
&\qquad \mathsf{then}\ \mathsf{map}\ (\lambda(x : \mathsf{int}) \to x * 5)\ xs \\
&\qquad \mathsf{else}\ \mathsf{map}\ (\lambda(x : \mathsf{int}) \to x * 20)\ xs \\
&\quad)\ xss\ bs \\
&\mathsf{in}\ res
\end{aligned}
$$

Here $xss : [n][m]\mathsf{int}$ and $bs : [n]\mathsf{bool}$. The condition b is taken from the array bs and is therefore variant with respect to the outer map. For example, consider the concrete inputs:

$$xss = [[0, 1], [2, 3], [4, 5]] \qquad bs = [\mathsf{true}, \mathsf{false}, \mathsf{true}]$$

Then the first and third outer iterations take the then-branch, while the second takes the else-branch, and the result is:

$$[[0, 5], [40, 60], [20, 25]]$$

The idea is to evaluate each branch only on the iterations that selected it, and then write the results back to their original positions. This proceeds in three steps: partition the outer indices by branch, split the variant inputs accordingly, and merge the branch results back afterwards. This follows the branch-packing approach used in classical full flattening [3].

Partitioning and Splitting

We first compute which outer iterations take each branch, by partitioning the outer indices according to the condition:

$$
\begin{aligned}
&\mathsf{let}\ inds : [n]\mathsf{int} = \mathsf{iota}\ n \\
&\mathsf{let}\ (is_T, is_F) = \mathsf{partition}\ (\lambda i \to bs[i])\ inds
\end{aligned}
$$

For the example above, this gives $is_T = [0, 2]$ and $is_F = [1]$.

Next, we split the variant inputs used by the branches. In this example, the only such input is xss, because each branch uses the current row xs:

$$
\begin{aligned}
&\mathsf{let}\ xss_T = \mathsf{split}\ xss\ is_T \\
&\mathsf{let}\ xss_F = \mathsf{split}\ xss\ is_F
\end{aligned}
$$

For the concrete input, this gives $xss_T = [[0, 1], [4, 5]]$ and $xss_F = [[2, 3]]$.

Evaluating the Branches

After splitting the inputs, each branch can be flattened independently. The then-branch is evaluated only on the inputs in xss_T:

$$\text{let } res_T : [2][m]\text{int} = \text{segmap}\langle q < 2, j < m\rangle\ xss_T[q, j] * 5$$

and the else-branch only on the inputs in xss_F:

$$\text{let } res_F : [1][m]\text{int} = \text{segmap}\langle q < 1, j < m\rangle\ xss_F[q, j] * 20.$$

For the example, this gives $res_T = [[0, 5], [20, 25]]$ and $res_F = [[40, 60]]$.

Merging the Results

Finally, the branch results are merged back into their original positions:

$$\begin{aligned}&\text{let } blank : [n][m]\text{int} = \text{replicate } n\ (\text{replicate } m\ 0)\\&\text{let } res'_T = \text{merge } blank\ is_T\ res_T\\&\text{let } res = \text{merge } res'_T\ is_F\ res_F\end{aligned}$$

For the concrete example, the first merge writes the then-results back to positions 0 and 2, giving $res'_T = [[0, 5], [0, 0], [20, 25]]$, and the second merge writes the else-result back to position 1, giving $res = [[0, 5], [40, 60], [20, 25]]$.

Lowering Split and Merge

We now show how split and merge are lowered when their inputs are regular arrays.

split is just indexing by the selected branch indices. For a regular two-dimensional input $xss : [n][m]\text{int}$, splitting by an index array $is : [r]\text{int}$ is lowered as:

$$\text{split } xss\ is = \text{segmap}\langle q < r, j < m\rangle\ xss[is[q], j]$$

merge is lowered to a scatter. Suppose $rows : [r][m]\text{int}$ contains the branch results and $is : [r]\text{int}$ contains their original outer positions. The scatter uses is to place each result back at its corresponding position in the destination:

$$\text{merge } dest\ is\ rows = \text{scatter } dest\ is\ rows.$$

We see that even when the condition is variant, the compiler can still exploit the parallelism inside the conditional branches. The cost is that the outer iteration space must first be partitioned by branch, the variant inputs must be split accordingly, and the branch results must be merged back afterwards.

Split and Merge for Irregular Data

The previous example used a regular variant input, so splitting and merging were just ordinary indexing and scattering of rows. The same idea also works when the variant input is irregular. The only difference is that we must split and merge both the segment metadata and the flat data array.

Consider this irregular array:

$$xss = [\ [5,6],\ [4,7,4],\ [5]\ ]$$

whose flat representation is:

$$xss_S = [2,3,1] \qquad xss_O = [0,2,5]$$

$$xss_F = [T,F,T,F,F,T] \qquad xss_D = [5,6,4,7,4,5]$$

Assume that the conditional has produced the same branch index arrays as before, $is_T = [0,2]$ and $is_F = [1]$, so the true branch should receive segments 0 and 2, and the false branch segment 1. Splitting the true input gives:

$$xss^T_S = [2,1] \qquad xss^T_O = [0,2]$$

$$xss^T_F = [T,F,T] \qquad xss^T_D = [5,6,5]$$

and splitting the false input gives:

$$xss^F_S = [3] \qquad xss^F_O = [0]$$

$$xss^F_F = [T,F,F] \qquad xss^F_D = [4,7,4]$$

We now describe how the true split is computed. The false split is analogous. First, the new segment sizes are obtained by indexing the old segment-size array with the selected branch indices:

$$xss^T_S = \mathsf{segmap}\langle q < |is_T| \rangle\ xss_S[is_T[q]]$$

From these segment sizes, we compute the new offsets, total flat size, and flags as usual:

$$xss^T_O = \mathsf{scan}_{exc}\ (+)\ 0\ xss^T_S \qquad m_T = \mathsf{sum}\ xss^T_S \qquad xss^T_F = \mathsf{gen\_flags}\ m_T\ xss^T_S\ xss^T_O$$

The only non-trivial part is constructing the new flat data array xss^T_D. For this, we derive the segment-index and intra-segment-index arrays from xss^T_S:

$$II^T_1 = [0,0,1] \qquad II^T_2 = [0,1,0]$$

For each flat position p in the new data array, $II^T_1[p]$ tells us which new segment we are in, and $II^T_2[p]$ tells us the local index inside that segment. The new segment

index is then mapped back to an old segment index through is_T, so the data array is constructed as:

$$\begin{aligned}&\textsf{let } xss_D^T : [m_T]\textsf{int} = \textsf{segmap}\langle p < m_T\rangle\\&\quad\textsf{let } q = II_1^T[p]\\&\quad\textsf{let } local = II_2^T[p]\\&\quad\textsf{let } old\_seg = is_T[q]\\&\quad\textsf{let } old\_off = xss_O[old\_seg]\\&\quad\textsf{in } xss_D[old\_off + local]\end{aligned}$$

For the concrete input, this gives $xss_D^T = [5, 6, 5]$, as expected.

After the branches have been evaluated, their results must be written back to the original outer positions. For irregular results, this means first constructing the final segment-size array and then scattering the branch data into the correct flat positions. Suppose the true branch produces an irregular result $\langle ys_S^T, ys_F^T, ys_O^T, ys_D^T\rangle$ for the indices is_T, and the false branch produces $\langle ys_S^F, ys_F^F, ys_O^F, ys_D^F\rangle$ for the indices is_F. We first create a blank segment-size array of length n and scatter the branch segment sizes back to their original positions:

$$\begin{aligned}&\textsf{let } S_0 : [n]\textsf{int} = \textsf{replicate } n\ 0\\&\textsf{let } S_1 = \textsf{scatter } S_0\ is_T\ ys_S^T\\&\textsf{let } S = \textsf{scatter } S_1\ is_F\ ys_S^F\end{aligned}$$

From this final segment-size array, we compute the final offsets, total flat size, and flags:

$$O = \textsf{scan}_{exc}\ (+)\ 0\ S, \qquad m = \textsf{sum } S, \qquad F = \textsf{gen\_flags } m\ S\ O$$

It remains to construct the final flat data array D. We start with a blank flat array of length m, and then each branch result is scattered into the positions determined by the final offsets. For the true branch, we derive II_1^T and II_2^T from ys_S^T, and for each flat element p of ys_D^T the destination index is computed by:

$$\begin{aligned}&\textsf{let } idxs_T : [|ys_D^T|]\textsf{int} = \textsf{segmap}\langle p < |ys_D^T|\rangle\\&\quad\textsf{let } q = II_1^T[p]\\&\quad\textsf{let } local = II_2^T[p]\\&\quad\textsf{let } dst\_seg = is_T[q]\\&\quad\textsf{let } dst\_off = O[dst\_seg]\\&\quad\textsf{in } dst\_off + local\end{aligned}$$

The true-branch data is then written by scattering ys_D^T to these indices:

$$\textsf{scatter } D\ idxs_T\ ys_D^T.$$

The false branch is handled in the same way, using is_F, ys_S^F, and ys_D^F.

5.2 Rearrange

We now consider rearrange, which permutes the dimensions of an array. As with conditionals, the treatment depends on whether the operation is uniform with respect to the surrounding map nest. A rearrange is uniform if the array it operates on is uniform meaning that the size type of array it permutes is invariant in the map nest. A uniform rearrange can be handled almost for free, whereas a non-uniform one must operate on the irregular representation and is considerably more expensive. It is therefore important to detect the uniform case.

5.2.1 Uniform Rearrange

Consider a rearrange inside a map:

$$\mathsf{map}\ (\lambda(xs : [n][m]\mathsf{int}) \to \mathsf{rearrange}\langle 1, 0\rangle\ xs)\ (xss : [k][n][m]\mathsf{int})$$

The type of xs, namely $[n][m]$int, is invariant in the map nest, since the sizes n and m do not depend on the iteration. This is therefore a uniform rearrange. The collected value xss is a regular array of type $[k][n][m]$int, and the per-iteration value xs is represented directly by xss.

Because the array is regular, we do not need to operate on a flat representation at all. The whole map can be replaced by a single top-level rearrange on xss. The outer dimension of xss, which indexes the segments, must be left in place, and the permutation that applied to xs must be shifted to act on the inner dimensions. In this example, the inner permutation $\langle 1, 0\rangle$ becomes $\langle 0, 2, 1\rangle$ on xss:

$$\mathsf{map}\ (\lambda xs \to \mathsf{rearrange}\langle 1, 0\rangle\ xs)\ xss\ =\ \mathsf{rearrange}\langle 0, 2, 1\rangle\ xss$$

The leading 0 keeps the segment dimension fixed, and the remaining entries are the inner permutation with each index increased by one, to account for the segment dimension now sitting in front.

In general, suppose the surrounding map nest has segment-rank r, so that the first r dimensions of the collected array index the iterations of the nest, and the inner rearrange applies a permutation $perm$ to the remaining dimensions. Then the whole nest is equivalent to a single top-level rearrange with the permutation:

$$\langle 0, 1, \ldots, r-1\rangle\ +\!\!+\ \mathsf{map}\ (\lambda d \to d + r)\ perm$$

where $+\!\!+$ denotes concatenation of index sequences. That is, the identity on the segment dimensions followed by the inner permutation shifted by r. This is correct because the dimensions that belong to the segment structure are unchanged, and only the inner dimensions are permuted.

A uniform rearrange is thus represented by a single top-level rearrange. Such a rearrange is very efficient and at the level of the generated code it amounts to a change of indexing, and need not move any data.

5.2.2 Non-Uniform Rearrange

When the array permuted by the rearrange has a variant type in the map nest, the collected value is irregular, and the simple top-level rearrange no longer applies. Consider this example:

$$\mathsf{map}\ (\lambda(xs : [][]\mathsf{int}) \to \mathsf{rearrange}\langle 1, 0\rangle\ xs)\ (xss : [k][][]\mathsf{int})$$

Here we write the inner type loosely as $[][]\mathsf{int}$ to indicate that each segment is a two-dimensional array whose extents vary between iterations. The two inner dimensions are represented by the per-segment size arrays ns and ms, so the map effectively ranges over these sizes as well, binding the per-iteration extents n and m and giving each iteration a regular value $xs : [n][m]\mathsf{int}$:

$$\mathsf{map}\ (\lambda(n : \mathsf{int})\ (m : \mathsf{int})\ (xs : [n][m]\mathsf{int}) \to \mathsf{rearrange}\langle 1, 0\rangle\ xs)\ ns\ ms\ xss$$

Within a single iteration, then, xs is an ordinary regular array, and the irregularity lies only in how the segments are collected. Since xss is represented by $\langle xss_S, xss_F, xss_O, xss_D\rangle$, we must permute the elements within each segment individually while keeping the segment metadata intact. The segment sizes, offsets, and flags are unchanged, since the rearrange only permutes the elements inside each segment, and only the flat data array is rebuilt.

The flat data array is constructed by a segmap over the $|D|$ flat positions. At each position p it reads the segment index $j = II_1[p]$ and the local index $q = II_2[p]$ within that segment, looks up the segment offset $O[j]$ and the inner shape of segment j from the per-segment size arrays, and computes the source position by transposing the local index according to the permutation:

$$\begin{aligned}
&\mathsf{let}\ D' : [|D|]\mathsf{int} = \mathsf{segmap}\langle p < |D|\rangle \\
&\quad \mathsf{let}\ j = II_1[p] \\
&\quad \mathsf{let}\ q = II_2[p] \\
&\quad \mathsf{let}\ o = O[j] \\
&\quad \mathsf{let}\ dims = [\, ns[j],\ ms[j]\,] \\
&\quad \mathsf{let}\ q' = \mathsf{rearrangeFlat}\ perm\ dims\ q \\
&\quad \mathsf{in}\ D[o + q']
\end{aligned}$$

Here rearrangeFlat maps a target local flat index inside a segment to the source local flat index from which the value should be read. It does so by unflattening the index with the permuted shape, permuting the resulting multidimensional index back with the inverse permutation, and flattening it again with the original shape. The resulting flat data array D', together with the unchanged segment metadata, forms the rearranged irregular value.

Example. Consider the irregular array

$$xss = [\, [[1, 2, 3], [4, 5, 6]],\ [[7, 8]]\,]$$

whose first segment is a 2×3 array and whose second is a 1×2 array. Its flat representation is:

$$xsss_S = [6, 2] \qquad xss_O = [0, 6]$$
$$xss_F = [T, F, F, F, F, F, T, F] \qquad xss_D = [1, 2, 3, 4, 5, 6, 7, 8]$$

and the per-segment inner shapes are $ns = [2, 1]$ and $ms = [3, 2]$. Applying $\mathsf{rearrange}\langle 1, 0\rangle$ inside the map transposes each segment, so the logical result is:

$$[\,[[1, 4], [2, 5], [3, 6]],\ [[7], [8]]\,]$$

The segment metadata is unchanged. Only the flat data array is rebuilt. Using the index arrays:

$$II_1 = [0, 0, 0, 0, 0, 0, 1, 1] \qquad II_2 = [0, 1, 2, 3, 4, 5, 0, 1]$$

The segmap computes, for each flat position p, the source position within the segment. For example, position $p = 1$ lies in segment 0 at local index $q = 1$, which corresponds to the logical index $[0][1]$ in the transposed array. The reverse permutation of $\langle 1, 0\rangle$ is $\langle 1, 0\rangle$ itself, so permuting $[0][1]$ gives $[1][0]$, which corresponds to local flat index 3. This is the position we read from, so the value read is $xss_D[0 + 3] = 4$. Carrying this out for every position gives the new flat data array:

$$xss'_D = [1, 4, 2, 5, 3, 6, 7, 8]$$

which is exactly the flat representation of the transposed result above.

This is considerably more work than the uniform case. Where the uniform rearrange was a single top-level permutation that moves no data, the non-uniform rearrange constructs the index arrays, reads the per-segment shapes, and performs a per-element gather over the flat data array. This again illustrates why detecting the uniform case is crucial as the two cases have very different costs.

5.3 Functions

We now consider function calls. Suppose a function f is applied inside a map:

$$\mathsf{map}\ (\lambda x \to f\ x)\ xs$$

If f is parallel, then the call introduces nested parallelism that flattening must remove. As in previous work on flattening transformation [23], we replace the call by a call to a *lifted* version of the function, written $f^{\uparrow}$, that operates on all iterations of the map at once:

$$\mathsf{map}\ (\lambda x \to f\ x)\ xs \ \rightsquigarrow\ f^{\uparrow}\ xs$$

The lifted function takes the collected arguments of all iterations and returns the collected results. To illustrate, let us define f as follows:

$$f\ (xs : [m]\mathsf{int}) : [m]\mathsf{int} = \mathsf{map}\ (\lambda x \to x + 5)\ xs$$

and suppose f is called inside a map:

$$\mathsf{map}\ (\lambda xs \to f\ xs)\ xss$$

We first lift the definition of f, and then lift the function application.

5.3.1 Lifting the Function Definition

We add a new parameter w to the function, corresponding to the width of the map nest at which the function is called. We then lift each original parameter.

A scalar parameter is lifted to an array of length w. An array parameter is lifted to its irregular representation. Because a function has only a single lifted version, which must serve every call site, we assume the worst case, namely that the argument may be irregular. Each array parameter therefore becomes five parameters: an integer *num_data* that holds the number of elements in the flat data, the segment-size and segment-offset arrays, of length w, and the flat data and flag arrays, of length *num_data*.

Strictly speaking, the size m in the type of xs is itself an implicit parameter of f, and it is lifted together with the explicit ones. Like any scalar parameter, it becomes an array of length w holding the per-iteration sizes. However, we omit them here since types are considered implicit inputs.[1]

In our example, f has a single array parameter xs. After lifting, the parameter list becomes:

$$(w : \mathsf{int}) \quad (\mathit{num\_data} : \mathsf{int}) \quad (xs_S : [w]\mathsf{int}) \quad (xs_O : [w]\mathsf{int})$$
$$(xs_F : [\mathit{num\_data}]\mathsf{bool}) \quad (xs_D : [\mathit{num\_data}]\mathsf{int})$$

We then lift the body by flattening it with respect to a map of width w, using the rules of the previous chapters. We first distribute, then flatten the body. The body $\mathsf{map}\ (\lambda x \to x + 5)\ xs$ becomes a flat map over the data array and the lifted function returns the same segment metadata together with a new data array:

$$\mathsf{let}\ xs'_D = \mathsf{segmap}\langle p < \mathit{num\_data}\rangle\ xs_D[p] + 5$$

The lifted function $f^{\uparrow}$ returns $\langle \mathit{num\_data}, xs_S, xs_O, xs_F, xs'_D\rangle$. The segment metadata is unchanged, because adding 5 to each element does not change the shape.

5.3.2 Lifting the Function Call

At the call site, each argument must be lifted to match the parameters of the lifted function. Suppose the call occurs in a map of width k, so the new width parameter is $w = k$. A scalar argument is lifted to an array of length k. If the argument is free in the map nest, it is replicated across the k iterations. Otherwise, its distributed version is used directly.

An array argument is lifted to its irregular representation, and the length of its flat data array is passed as an additional argument. If the array argument is free in the map nest, its lifted representation is obtained by replicating the array across the k iterations. Otherwise, if its distributed version is already irregular, that representation is passed directly. If the array is regular, its irregular representation is generated.

Suppose in the example $xss = [[1, 2],\ [3, 4, 5],\ [6]]$ is represented by:

$$xss_S = [2, 3, 1] \qquad xss_O = [0, 2, 5]$$

[1] In the implementation, the implicit size parameters are lifted in the same way as ordinary scalar parameters.

$$xss_F = [T, F, T, F, F, T] \qquad xss_D = [1, 2, 3, 4, 5, 6]$$

with *num_data* = 6. The call map $(\lambda xs \to f\ xs)\ xss$ becomes:

$$f^{\uparrow}\ k\ num\_data\ xss_S\ xss_O\ xss_F\ xss_D$$

which runs the lifted body, and returns the data array with the unchanged metadata:

$$num\_data = 6$$

$$xss_S = [2, 3, 1] \qquad xss_O = [0, 2, 5]$$

$$xss_F = [T, F, T, F, F, T] \qquad xss'_D = [6, 7, 8, 9, 10, 11]$$

which is the flat representation of $[\,[6, 7],\ [8, 9, 10],\ [11]\,]$ exactly the result of applying f in each iteration of the original map.

5.3.3 Avoiding Unnecessary Lifting

It is worth noting that we only need to lift functions that are themselves parallel. If a function performs only scalar computation, lifting it would incur cost without providing any benefit as there is no exploitable parallelism inside the function. For example, given the following function f':

$$f'\ x = x + x$$

we do not lift f' since there is no parallel expression in its body. Then the function applications such as map $(\lambda x \to f'\ x)\ xs$ remains unchanged inside the map. This follows from the vectorisation avoidance introduced in Section 4.2.

5.3.4 Recursive Functions

Recursive functions can also be lifted. The key observation is that a recursive call inside the function body is just another function call. Consequently, when the body is lifted, the recursive call is rewritten as a call to the lifted version of the function. Consider the following function, which takes a scalar n and an array xs, and whose body contains a recursive call:

```
fun f ((n : int), (xs : [m]int)) : int =
  if n = 0 then 0 else
    let ys : [m]int = map (λ(x : int) → x + 1) xs
    let s : int = sum ys
    let n' : int = n − 1
    let r : int = f n' ys
    in s + r
```

Each call increments every element, sums the result, and recurses on the new array with $n - 1$.

We lift the definition of f as in Section 5.3.1. The scalar parameter n becomes an array ns of length w, and the array parameter xs becomes the components of its irregular representation. Before giving the lifted function, we note a small notational shift. The segmented operations were introduced as consuming nested values, as in segmented_sum yss. Here we instead write them as consuming the representation tuple directly, as in segmented_sum $\langle S, F, O, D\rangle$, which is exactly the form in which they are lowered in Section 3.6. Flattening the body with respect to a map of width w then gives the following lifted function:

```
fun f↑ ( (w : int), (num_data : int), (ns : [w]int),
         (xs_S : [w]int), (xs_O : [w]int),
         (xs_F : [num_data]bool), (xs_D : [num_data]int) ) : [w]int =
  if w = 0
    then replicate w 0
    else
      let (is_B, is_R) = partition (λj → ns[j] = 0) (iota w)
      let w_B = |is_B|
      let w_R = |is_R|
      let res_B : [w_B]int = replicate w_B 0
      let ns_R : [w_R]int = segmap⟨q < w_R⟩ ns[is_R[q]]
      let ⟨xs_S^R, xs_F^R, xs_O^R, xs_D^R⟩ = split ⟨xs_S, xs_F, xs_O, xs_D⟩ is_R
      let m_R = |xs_D^R|
      let ys_D : [m_R]int = segmap⟨p < m_R⟩ xs_D^R[p] + 1
      let ss : [w_R]int = segmented_sum ⟨xs_S^R, xs_F^R, xs_O^R, ys_D⟩
      let ns' : [w_R]int = segmap⟨q < w_R⟩ ns_R[q] − 1
      let rs : [w_R]int = f↑ w_R m_R ns' xs_S^R xs_O^R xs_F^R ys_D
      let res_R : [w_R]int = segmap⟨q < w_R⟩ ss[q] + rs[q]
      let blank : [w]int = replicate w 0
      let res' = scatter blank is_B res_B
      let res = scatter res' is_R res_R
      in res
```

Apart from the width-zero guard, the body uses the machinery introduced previously. Since the condition $n = 0$ is variant, the iterations are partitioned into base-case indices is_B and recursive-case indices is_R, and the corresponding inputs are split. The base branch produces one zero per base-case iteration. In the recursive branch, the inner map is flattened over the data array, and the sum becomes a segmented_sum. The two branch results are then merged by scattering them back to their original positions.

The important point is the recursive call. When the body is lifted, $f\ n'\ ys$ is rewritten as a call to $f^{\uparrow}$. The scalar argument n' is already available as the array ns', and the array argument ys is already in its irregular representation $\langle xs_S^R, xs_F^R, xs_O^R, ys_D\rangle$, so both are passed directly. The lifted function therefore calls itself once on the collected inputs of all iterations that take the recursive branch.

The guard $w = 0$ prevents infinite recursion after every iteration has reached its base case. At that point, is_R is empty and the recursive call has width zero. The flat and segmented operations naturally produce empty results on such inputs, but an unguarded call to $f^{\uparrow}$ would repeatedly call itself with width zero. The guard therefore returns an empty array immediately. Its element value is irrelevant because a width-zero result contains no elements.

Finally, we note that this treatment is not exercised by our implementation, since Futhark deliberately omits recursion (Section 2.4). We include it because it shows that lifting is not limited to the non-recursive functions.

Chapter 6

Implementation

In the previous chapters we described our flattening transformation abstractly, with a small source and target language. This chapter describes how the flattening transformation is implemented as a middle-end pass in the Futhark compiler. The gap between the abstract source language and Futhark is large, as Futhark is a real, usable language with many more constructs that we need to handle. Moreover, we must be careful about many details to achieve good performance.[1]

We organise the chapter to mirror the conceptual structure of the earlier chapters. We first give an overview of the pass and the modules that comprise it (Section 6.1). We then describe the central data structures: the representations of irregular and distributed values (Section 6.2). The following sections follow the pipeline a program flows through: preprocessing (Section 6.3), distribution (Section 6.4), and then the flattening rules proper for basic operations (Section 6.5), SOACs (Section 6.6), and control-flow and function constructs (Section 6.7). Finally, Section 6.8 describes how incremental flattening is integrated.

6.1 Overview

As mentioned, the flattening transformation is implemented as a middle-end pass that transforms a program from the SOACS IR, which supports nested parallelism, into the GPU IR, in which all parallelism is flat. The pass looks for instances of top-level parallelism and applies flattening to them. In Futhark, reduce and scan are fused with a map and represented as redomap and maposcanomap. Together with map, these are the entry points into flattening. When such a statement is encountered, its map body is distributed and flattened, and several semantically equivalent versions are generated as described in Section 6.8. A loop, match, or with_acc is instead traversed structurally, so that any parallel construct nested inside it becomes an entry point in turn. A statement that is scalar is simply translated from the SOACS to the GPU representation unchanged. Lifted versions of parallel functions and the segmented builtins used by the transformation are generated on demand by fixed-point iteration.

[1] The implementation is available at https://github.com/diku-dk/futhark/tree/flattening.

The whole pass runs in a builder monad, FlattenM, which constructs GPU code and threads the state the transformation needs. The implementation is organised into roughly one module per major language construct or group of related constructs.

Finally, we note that the pass does not worry about small inefficiencies in the code it emits. Instead, it relies on a subsequent simplification pass over the generated GPU IR to perform local rewrites such as copy propagation, constant folding, and the simplification of indexing into replicated and rearranged arrays, as discussed in Section 4.3.4.

6.2 Representations

This section describes how we represent values and distributed statements. We first describe the representation of irregular and regular values, and then the representation of the results of distributed statements.

6.2.1 Representing Irregular Arrays

Chapter 3 introduced the four-array representation of an irregular value, the tuple $\langle S, F, O, D \rangle$ of segment sizes, flags, offsets, and flat data. In the implementation this becomes the record `IrregularRep`, holding the names of the segment-size array, flag array, offset array, and data array. The record additionally carries an `IrregularKind`, which is either `Dense` or `Replicated`:

```
data IrregularKind
  = Dense
  | Replicated

data IrregularRep = IrregularRep
  { irregularS :: VName,
    irregularF :: VName,
    irregularO :: VName,
    irregularD :: VName,
    irregularK :: IrregularKind
  }
```

`IrregularKind` is used for the implementation of the replication avoidance optimisation of Section 4.1. A `Dense` representation is the ordinary packed representation, in which the offset array is the exclusive prefix sum of the segment sizes and the data array is laid out contiguously. A `Replicated` representation is one whose offsets instead point back into an *unreplicated* data array, exactly the representation described in Section 4.1, so that a replicated irregular value can be read without copying its data. A regular replicated array currently reuses this irregular representation, carrying the full four-array structure it does not need. A dedicated regular representation would avoid that redundancy and could reuse the segment-index array directly, which we leave to future work. A `Replicated` representation is materialised into a `Dense` one only when a later operation requires the contiguous layout.

6.2.2 Representing Distributed Values

We use the `ResRep` type to describe the representation of a result after distribution. A distributed value can either be regular or irregular. The type `ResRep` is therefore either `Regular`, holding a single array name, or `Irregular`, holding an `IrregularRep`.

As described later in Section 6.4, each distributed statement is assigned a `ResTag` that identifies its result. The environment `DistEnv` maps these abstract result tags to their representations.

```
data ResRep
  = Regular VName
  | Irregular IrregularRep

newtype DistEnv = DistEnv { distResMap :: M.Map ResTag ResRep }
```

Flattening a distributed statement consists of looking up the representations of its inputs in this environment, generating the corresponding flattened code, and then inserting the representations of its results back into the environment.

6.3 Preprocessing

Before flattening, code is rewritten into the normal form the rest of the pass expects by the `PreProcess` module. Three rewrites are performed. First, stream constructs are sequentialised over the whole array, since the flattening rules do not handle streams. Second, a scan or reduce whose operator is itself just a map is interchanged with that map, turning a scan-of-map into a map-of-scan and likewise for reductions, which exposes the parallelism the inner map has to offer. Third, a fused combination of scan, reduce, and map (a Screma) whose form is not one of the recognised fused forms, such as a redomap, is *dissected* into a composition of those recognised forms.

An important subtlety is that we only preprocess code that we are immediately about to flatten. Preprocessing recurses into the bodies of compound constructs such as sequential loops and conditionals, but not into the body of a Screma, since whether a map body needs normalising depends on the version produced by incremental flattening. A map body is therefore preprocessed only when producing the fully flattened version, immediately before it is flattened.

6.4 Distribution

Distribution is implemented in the `Distribute` module. It corresponds directly to Section 3.3 and turns a map nest into a *distributed* representation, a value of type `Distributed`, consisting of a list of distributed statements together with a description of how the results of the original map nest are recovered.

Each input to the distributed nest is a `DistInput`. An input is either *free*, meaning it is one of the arrays the original nest maps over and is therefore necessarily regular,

or it is internal, in which case it carries a `ResTag` identifying the result of an earlier distributed statement:

```
data DistInput
  = DistInputFree VName Type
  | DistInput ResTag Type
```

Each result of the nest is a `DistResult`, pairing a fresh `ResTag` with a `DistType` and the variable's result name in the original map nest:

```
data DistResult = DistResult
  { distResTag :: ResTag,
    distResType :: DistType,
    distResName :: VName
  }
```

The `DistType` is the implementation's record of uniformity. It splits the type of a produced value into a leading regular part (the segment dimensions of the nest), a count of irregular dimensions stacked on top, and an inner regular element type. The split is computed by `splitIrregDims`, which walks the dimensions of a type from the innermost outward and classifies each as regular if its size is bound outside the nest and irregular otherwise. A distributed result is regular, tested by `isRegularDistResult`, when it has zero irregular dimensions. This single predicate tells us whether the resulting distributed value will be regular or irregular.

Distributed statements are represented by the type `DistStm`, which contains three fields. The field `distStmInputs` stores the distributed inputs as pairs of the original input variable name and its `DistInput` representation. The field `distStmResult` stores the distributed results as a list of `DistResult` values. Finally, `distStmBody` contains the body of the distributed statement:

```
type DistInputs = [(VName, DistInput)]

data DistStm = DistStm
  { distStmInputs :: DistInputs,
    distStmResult :: [DistResult],
    distStmBody :: DistBody
  }
```

We process the statements of the map nest one by one to create `DistStm`s, and each statement in the original map nest corresponds to one `DistStm`. One important thing to note here is that `distStmInputs` corresponds to inputs to the statement that are variant in the map nest. They are either inputs to the original map nest or results created by a preceding `DistStm`. Therefore, when we want to check whether something is variant or not, we can just check whether it belongs to the `distStmInputs` or not.

The body of a distributed statement is a `DistBody`, which has two forms:

```
data DistBody
  = ParallelStm (Stm SOACS)
  | ScalarStm [Stm SOACS]
```

The two forms of `DistBody` reflect vectorisation avoidance (Section 4.2). A `DistStm` contains a `DistBody` that is either a `ParallelStm`, a single statement that either contains parallelism or produces a non-uniform array, or a `ScalarStm`, a group of purely scalar statements that are kept together and will be flattened as a single ordinary segmap.

The grouping is performed by `classifyStms` after the statements have been laid out. It scans the statement list, accumulating runs of scalar statements and breaking them whenever a parallel statement is encountered. A statement is parallel, as decided by `isParallelDistStm`, if it contains a parallel operation or if any of its results is non-uniform. Each maximal scalar run is merged by `mergeGroup` into a single `ScalarStm`, retaining only the inputs that come from outside the group and only the results that are used later or that form part of the body result. Consecutive scalar statements are thus kept together inside one kernel rather than each producing its own intermediate array.

6.5 Flattening Basic Operations

The rules for flattening BasicOps live in the `BasicOp` module, in the dispatch function `transformDistBasicOp`. Each case uses the representations of its inputs and inserts the representation of its result into the environment.

The purely scalar operations (BinOp, CmpOp, ConvOp, UnOp, Assert, and Opaque) are delegated to the scalar case, which simply emits them inside a segmap over the segment space. Because vectorisation avoidance has already grouped consecutive scalar statements during distribution, this is rarely reached for a single isolated scalar operation.

The array-producing operations follow the abstract rules closely, and most share the same shape. A uniform case that operates on a regular array of the segment shape, and an irregular case built on segmented builtins and the segment and intra-segment index arrays. reshape is a regular reshape in the uniform case and otherwise simply re-tags the existing representation, since it moves no data, and scratch produces a regular array when all its dimensions are invariant and an irregular one otherwise. The operations iota, replicate, index, flat_index, concat, update, flat_update, and rearrange all follow the same pattern, dictated by the uniformity of the operation, just as we saw for rearrange in Section 5.2. We first examine whether the operation is uniform, using `isRegularDistResult` to check that its result is regular and `isVariant` to check that its size types are invariant. If so, the operation is handled by lifting its inputs to regular arrays of the segment shape and emitting the corresponding regular operation directly, which is usually cheap. Otherwise the operation is handled on the irregular representation, which is costly. This is the implementation's realisation of the uniform/non-uniform distinction that recurs throughout Chapters 4 and 5.

One thing to note is that update_acc reaches the scalar case in `BasicOp`. This is because, by the time it is flattened here, the `WithAcc` module (Section 6.7) has already rewritten its index to address the flat data array of the irregular representation, so the operation is genuinely scalar at this point.

6.6 Flattening SOACs

The flattening of SOACs is implemented in the `SOAC` module. The entry point, `transformScrema`, consists of guarded cases that classify the SOAC and dispatches to the appropriate strategy.

6.6.1 Inner Maps

An inner map is the heart of the transformation, and its handling distinguishes the two modes that Chapters 3 and 4 treat separately. The function `transformInnerMap` preprocesses the map body and then chooses, based on whether the map width is variant, between a `MultiDim` and a `SingleDim` mode. In `MultiDim` mode the width is invariant, so the nest is regular and is flattened into an additional regular dimension of the segment space, exactly the uniform nested map of Section 4.3. In `SingleDim` mode the width is variant. The inner map widths are gathered into a size array, the offset and segment-index arrays are built from it, and the inner map becomes a flat segmap over the sum of the map widths across the segments, as in Section 3.7.

6.6.2 Reductions, Scans and Histograms

A redomap whose width is invariant and whose results are all regular is a uniform reduction and is lowered, as in Section 4.4, to a regular segmented reduction over the segment index space. Otherwise it is non-uniform and is lowered using a segmented scan over the flat data array, similar to the lowering of Section 3.6.4. If the map part of the redomap itself contains parallelism, the map is extracted and flattened separately from the reduction. In the uniform case we additionally keep the unsplit redomap as an alternative version that does not exploit the inner parallelism. A maposcanomap is handled in the same way. Similarly, a hist is lowered to a uniform segmented histogram when its inputs are uniformly shaped and its width is invariant, and is otherwise rewritten as a with_acc, which is then handled by the ordinary flattening rules for accumulators.

One restriction is that the reduction, scan, and histogram operators may not have variant free variables. This is a limitation of the target representation. The operator's lambda cannot carry a free variable that we would need to read per-segment. SOACs whose operators violate this restriction, which are rare in practice, are handled through the sequentialising fallback instead.

6.7 Flattening Remaining Constructs

The implementation of this part also closely follows Chapter 5. Apart from function calls, we distinguish between the uniform and non-uniform cases and flatten accordingly.

6.7.1 Conditionals

Conditionals are flattened in the `Match` module, with the variant and invariant cases split according to whether any scrutinee is variant, just as in Section 5.1. An invariant conditional is interchanged with the surrounding nest, as in Section 5.1.1, so that the branch is chosen once and each branch is flattened as an ordinary nested computation. A variant conditional implements the partition–split–merge strategy of Section 5.1.2. The outer indices are partitioned by which branch they select, the variant free variables used in each branch are split, each branch is flattened on its share of the inputs, and the branch results are scattered back into their original positions.

6.7.2 Loops

Loops are flattened in the `Loop` module, similarly to conditionals. The essential difference from a conditional is that a loop has loop-carried parameters, and a loop-carried array may change shape from iteration to iteration, so its representation must be chosen with care. The function `liftLoopParam` lifts each loop parameter together with its initial value, deciding via `needsIrregular` whether the parameter requires an irregular representation. A parameter needs the irregular representation when any of its dimensions is either a loop-parameter name or variant in the surrounding nest. Otherwise it can be lifted to a regular array with the segment dimensions prepended, and scalar parameters are lifted to regular arrays of the segment shape.

This has an interesting relation to the function lifting described in Section 6.7.4. There, every array is lifted to its irregular form, because a function has only a single lifted version and so must assume the general case. In a loop there is no such constraint, so we can make the more precise choice and avoid representing as irregular a parameter that can stay regular.

6.7.3 Accumulators

Accumulators are Futhark's way of expressing in-place, scatter-like updates while keeping the program safe to parallelise. A with_acc runs a lambda that has access to one or more accumulators, and an update_acc inside that lambda accumulates a value into an accumulator at a given index. Their flattening is implemented in the `WithAcc` module. The basic idea of flattening them is interchange. The with_acc is interchanged with the surrounding map nest, so that a single with_acc operates over the whole nest rather than once per iteration, and its body is distributed and flattened like any other nest. The index of every update_acc is then rewritten so that it addresses the flattened accumulator correctly. In the non-uniform case, the accumulator operates over the flat data array and the index becomes the segment offset plus the local index. In the uniform case, the index is extended with the segment dimensions.

6.7.4 Functions

Function calls in a flattened context are handled by lifting, exactly as described in Section 5.3. The lifted version of a function takes an additional width parameter w. Scalar parameters become arrays of length w, and each array parameter becomes the components of its irregular representation. Because a single lifted version must serve every call site, the boundary assumes the worst case. Every array argument is therefore lifted to its irregular representation at the call site, with regular arguments promoted accordingly.

Lifted functions are generated on demand, so only those actually needed are included in the resulting program. A call is redirected to a lifted callee only when the callee contains parallelism, as determined by a whole-program analysis computed up front. Calls to purely scalar functions remain unchanged, avoiding unnecessary lifting as described in Section 5.3.3.

6.8 Integrating Incremental Flattening

Full flattening exploits all available parallelism, which, as Section 4.5 argues, is not always desirable on real hardware. The implementation therefore generates several semantically equivalent versions of each top-level and versionable map and defers the choice between them to run time. The implementation lives in the `Incremental` and `Intrablock` modules. Three versions are produced, corresponding to those of Section 4.5: a fully flattened version, an outer-only version that parallelises the outer level and sequentialises the body, and an intra-block version that assigns one block to each outer iteration. Whether a particular version is generated is governed by a set of predicates. `worthSequentialising` and `worthIntrablock` score the body to decide whether the outer-only and intra-block versions are worth generating, using simple heuristics that weight nested parallelism. Futhark also has a set of user-supplied attributes (`only_intra`, `no_outer`, `sequential_inner`, `sequential`, and so on) that can force or suppress particular versions.

Generating multiple versions is not always possible. If the parallelism of the current map is irregular, the only available option is to fully flatten the map, and we can therefore generate only one version. Moreover, if the map body contains deeper irregular parallelism, we cannot generate the intra-block version because of technical limitations. In the current implementation, when we encounter a map one level deeper in the body, we fully flatten it.

One implementation limitation of generating multi-version code concerns function calls. If the map body contains a call to a parallel function, we cannot use the outer-only version, since that would leave unflattened nested parallelism inside the call. The intra-block version is likewise unavailable. The reason is a mismatch in the GPU level hierarchy. An intra-block kernel runs its body at the in-block level (`SegThreadInBlock`), which is only meaningful inside a surrounding block-level operation (`SegBlock`) that establishes the block context, as described in Section 4.5.4. A lifted function would contain in-block segmaps, but at the call site there is no

enclosing `SegBlock` for them to belong to, so the lifted function is not well-defined there. In both cases we therefore fall back to full flattening. Since almost all parallel functions in Futhark are currently inlined, this is not a significant limitation in practice.

Chapter 7

Evaluation

This chapter evaluates the flattening pass from four perspectives. First we check that it is correct, in the sense that it preserves the semantics of the programs it transforms (Section 7.1). Second we show that it improves the compiler's expressiveness, allowing the GPU backend to accept programs it previously rejected (Section 7.2). Third we evaluate performance, confirming that supporting irregular parallelism does not cost the performance the existing flattening achieved on regular programs, and demonstrating the performance it unlocks on irregular programs (Section 7.3). Finally, we measure the pass's compile-time overhead (Section 7.4).

7.1 Correctness

Flattening is a program transformation, and it is crucial that it preserves the semantics of the original program. Our correctness argument is empirical rather than formal. The flattening transformation has not been formalised, so we do not prove that it preserves semantics and instead rely on extensive testing to give us confidence that the pass produces correct results. Futhark comes with a large test suite whose programs are easy to run and check with the `futhark test` tool, which compiles each program and compares its output against expected values. We reuse this suite as our primary correctness check.

We ran the full suite of 2544 tests through our pass using the `opencl` backend, and all tests passed.[1] Since the suite covers all major parts of the Futhark language, this gives us confidence that our transformation preserves program semantics across the language.

Moreover, we tested the programs in the Futhark benchmark suite [24], which is used in Section 7.3.1. These programs are larger and more realistic than the test-suite programs, providing additional evidence that the transformation preserves program semantics.

The existing suite was, however, not written with irregular parallelism in mind, and most of its programs contain only regular parallelism. To cover this, we added

[1] The test suite is available at https://github.com/diku-dk/futhark/tree/flattening/tests.

tests that specifically target irregular parallelism.[2] These tests were derived from the flattening rules themselves. Each program construct handled by the pass can occur in several contexts, depending, for example, on whether its operands are invariant or variant to the enclosing map and whether the arrays involved are regular or irregular. The pass applies a different rule in each case. We therefore wrote tests aiming to cover each construct across the contexts in which it can occur. Some tests also contain larger compositions of constructs in which the interaction between several rules is tested. All these tests pass, which gives us confidence in the parts of the pass that the existing suite does not exercise.

7.2 Expressiveness

One of our goals, as discussed in Section 2.5, is to let the Futhark GPU backend accept more programs. Some programs that type-check are rejected by the existing backend because it cannot statically determine the memory each iteration needs. One example is a loop-carried array whose size grows as the loop runs, so that its final size is not known on entry to the surrounding map, as we already saw in Section 2.5:

```
map (\n ->
       let res =
         loop arr = [1]
         while length arr < n do
           arr ++ arr
       in i32.sum res)
    ns
```

Under the existing compiler this program type-checks but is rejected by the GPU backend. The length of `res` is discovered only as the loop runs, and since GPU memory must be allocated before a kernel is launched, the compiler cannot pre-compute a per-iteration size with which to lay the results out contiguously. The programmer is left to restructure the program by hand or abandon the GPU backend.

With our transformation, the loop is no longer run independently inside each map iteration. Instead, flattening lifts the loop to operate over all outer map elements at once. Effectively, we interchange the loop and the map. The loop becomes host-side sequential control flow around parallel kernels, while the loop body performs irregular parallel operations over the active segments. Since this outer loop is executed by the host in the GPU backend, the total size of the irregular data array may change from one loop iteration to the next. The changing size is therefore no longer a per-thread allocation problem inside a kernel, but a change in the flat irregular representation between kernel launches. Consequently, the program can be compiled by the GPU backend.

More generally, we are aware of no Futhark program that our pass fails to compile for the GPU backend. With our transformation, the GPU backends can compile every valid Futhark program (modulo compiler bugs). Where the backend previously

[2] The additional tests are located in the `tests/flattening` subdirectory.

rejected programs whose memory requirements it could not determine statically, it now accepts the full language.

7.3 Performance

We evaluate performance in two ways. First, we show that on regular workloads our pass matches the performance of the existing flattening, so that supporting irregular parallelism does not cost the performance Futhark already achieved. Second, we use sparse-matrix–vector multiplication as a case study of irregular parallelism, comparing the natural nested implementation, which the existing compiler could not exploit, against a flattened version written with the flattening-by-expansion library.

7.3.1 Futhark Benchmark Suite

To check that we do not regress on regular workloads, we use the Futhark benchmark suite [24], run with `futhark bench`, and compare our pass against the existing `ExtractKernels` pipeline on the same benchmarks and datasets. We exclude the micro-benchmarks. We use the `flattening` branch of the benchmark suite. Its only relevant difference from the main branch is that a few benchmarks, which were originally written to exploit specific behaviour of `ExtractKernels` by provoking it into producing particular intra-block code without using attributes, were modified slightly to instead request that code explicitly via attributes such as `#[sequential]`. We return to this point below.

All measurements in this section were taken on an NVIDIA L40S (48 GB, PCIe) with CUDA 12.0, using the CUDA backend. In total we measure 523 datasets across 96 entry points. No tuning was performed for either compiler, so all runs use the default thresholds.

Figure 7.1 summarises the results. Each bar shows the speedup of our pass over the existing compiler on the entry point's most impactful dataset, that is, the dataset with the largest $|\log_2|$ of the speedup, so that the dataset on which behaviour deviates most, in either direction, is the one displayed.

The overall picture is that performance is largely preserved. Across all 523 datasets, the geometric mean speedup is 0.99 and the median is 1.00. Moreover, only 9 of the 96 entry points regress by more than 10%. This indicates that supporting irregular parallelism introduces little or no overhead for most programs. A handful of benchmarks even improve, most notably the `gmm` benchmarks (6.83× for the Jacobian and 3.41× for the objective), Poseidon (1.89×), and the PBBS `merge_sort` and `sort_dedup_int` entry points (1.88–1.93×).

A small number of benchmarks show large regressions. One identifiable cause is threshold selection. At run time the fully flattened version is chosen even though the outer-only version would have been the better choice. This is a threshold-selection issue and we expect tuning the thresholds to remove these regressions.

A separate limitation concerns the intra-block version. In some programs, such as `lud`, the existing compiler executes the nest as an intra-block kernel, but our pass

Figure 7.1: Speedup of our pass over the existing compiler for each benchmark entry point, measured on the entry point's most impactful dataset (the dataset with the largest $|\log_2(\text{speedup})|$). The logarithmic axis spans $1/8$ to 8. Grey bars indicate results within $\pm 10\%$, blue bars indicate improvements, and red bars indicate regressions.

cannot, because our flattening introduces irregular parallelism inside the nest, and as discussed in Section 6.8, the intra-block version is not generated in that case. Losing the intra-block version means losing the benefit of using shared memory, which is what these benchmarks rely on for performance. On the unmodified benchmark this results in a dramatic regression, with `lud` running approximately 1000× slower. In the numbers reported here this is masked by the benchmark modifications described above, since with the inner parallelism explicitly marked `#[sequential]` the nest is regular and the intra-block version is recovered. Without such annotations the problem remains, and we leave a solution as future work.

We also observe that several of the regressions shrink as the dataset grows. For example, `fft` regresses more on 128×128 than on 1024×1024. This is consistent with the hypothesis that the wrong version is sometimes selected at runtime, which will be resolved by tuning the incremental flattening threshold.

We note that there is still likely room to improve performance, specifically by using heuristics in vectorisation avoidance (Section 4.2) for classifying statements as parallel or scalar. For example, keeping a statement that is currently classified as parallel inside the surrounding kernel can sometimes lead to better performance.

7.3.2 Sparse-Matrix–Vector Multiplication

Full flattening lets us exploit all the parallelism in a program, where the existing flattening of the Futhark compiler falls short. We use sparse-matrix–vector multiplication (SpMV) on a compressed-sparse-row (CSR) matrix to demonstrate this. A CSR matrix stores an array of values, an array giving the column of each value, and an array of row offsets marking where each row begins. Each row can be processed independently, but rows have different numbers of non-zero entries, so the parallelism across the entries is irregular. We can define the CSR type in Futhark:

```
type csr 't = { row_off: []i64, col_idx: []i64, vals: []t }
```

The natural way to write SpMV is as nested parallelism, an outer `map` over the rows with an inner `map` and `reduce` computing each row's dot product:

```
def smvm ({row_off, col_idx, vals}: csr f32)
         (v: []f32) : []f32 =
  map (\i ->
    let start = row_off[i]
    let end   = row_off[i+1]
    let diff  = end - start
    in reduce (+) 0f32
         (map (\j -> vals[j+start] * v[col_idx[j+start]])
              (iota diff)))
    (iota (length row_off - 1))
```

This program compiles on both the existing compiler and ours, but the difference is performance. The inner width `diff` potentially varies from row to row, so the inner parallelism is irregular, and the existing compiler cannot exploit it. It parallelises only

the outer map and runs each row's dot product sequentially. For matrices with few long rows this leaves most of the available parallelism unused. Our pass instead can flatten the irregular inner parallelism, and exploit all of the parallelism.

Before this work, recovering that inner parallelism meant flattening the irregularity by hand. Using the flattening-by-expansion library [22], the programmer supplies a function giving how many elements each row expands to and a function computing each expanded element, and passes them to an expansion combinator that performs the flattened reduction:

```
def smvm_expand ({row_off, col_idx, vals}: csr f32)
         (v: []f32) : []f32 =
  let rows =
    map (\i -> (i, row_off[i], row_off[i+1] - row_off[i]))
        (iota (length row_off - 1))
  let size row = row.2
  let get row i =
    let j = row.1 + i
    in vals[j] * v[col_idx[j]]
  in expand_outer_reduce size get (+) 0f32 rows
```

This recovers the inner parallelism, but at the cost of expressing the algorithm through the expansion interface rather than as a natural nested program.

We compare the performance of four configurations on the same datasets: the expansion-library version `smvm_expand` on the existing compiler, and the natural `smvm` on our pass, on the existing compiler's GPU backend, and, for reference, on the existing compiler's sequential C backend. We write `futhark` for our pass and `futhark-old` for the existing compiler. For our dataset, we use matrices with few rows but many columns, so that the outer map alone does not provide enough parallelism to saturate the GPU. The matrix is $10 \times 10{,}000{,}000$, and we vary its density from 1% to 20%. As density grows, each row does more inner work, so more of the total work lies in the inner parallelism that cannot be exploited by parallelising only the outer map. All measurements in this section were taken on an NVIDIA A100 (80 GB, PCIe) with CUDA 12.0 and an Intel Xeon Gold 6326 CPU running at 2.90GHz, using the CUDA backend, or the C backend for the sequential configuration. We report the arithmetic mean of at least 20 runs per dataset.

Table 7.1 shows the results. The natural nested program compiled by our pass is on average 296× faster (geometric mean) than the same program compiled by the existing compiler. The existing compiler exploits only the outer parallelism of width 10, which is far too little to occupy the GPU, and runs each row's dot product sequentially. The GPU code of existing compiler is even slower than simply compiling the same program to sequential CPU code with the C backend.

Compared with the expansion-library version, the flattened nested program achieves essentially identical performance. Across all densities, its geometric mean runtime is 0.3% higher than that of the expansion-library version. The natural program is therefore now a fully viable way to write high-performance irregular code. It matches the expansion library version while being written as an ordinary nested program,

Density (%)	`smvm_expand` `futhark-old`	`smvm` `futhark`	`smvm` `futhark-old`	`smvm` `futhark-old` (C)
1	318	336	66,549	7,956
2	436	452	127,858	14,981
3	577	575	184,168	23,698
4	712	710	235,962	24,500
5	855	861	284,069	27,288
10	1,504	1,491	482,945	38,422
15	2,129	2,056	648,175	40,651
20	2,775	2,718	798,621	40,928

Table 7.1: SpMV runtimes in microseconds (arithmetic mean of at least 20 runs) on a $10 \times 10{,}000{,}000$ CSR matrix at varying density, on an NVIDIA A100 with the CUDA backend. The first column is the expansion-library version, the second the natural nested version compiled by our pass, the third the same nested version compiled by the existing compiler, and the fourth the nested version compiled by the existing compiler's sequential C backend, run on the CPU.

whereas before this would have been unusably slow on the GPU.

7.4 Compile-Time Overhead

We measure how much compile time our pass adds relative to the existing compiler. We compiled the 69 programs in the Futhark benchmark suite (again excluding the micro-benchmarks) with the CUDA backend under both compilers. Measurements were taken with `hyperfine` [25], using one warm-up run followed by at least five timed runs per program, on an AMD EPYC 9334 machine with 32 GB of RAM.

Across the 69 programs, our pass increases compile time by 4.7% on average (geometric mean), with a median overhead of 1.2% (1.012×). The overhead is therefore modest. One of the main sources of additional compile time is the generation of lifted functions, so we would expect programs with many functions requiring lifting to incur a larger overhead. The programs in the Futhark benchmark suite do not, which is consistent with the small gap we observe, since inlining leaves few functions to be lifted.

Chapter 8

Conclusion

This thesis has developed a full flattening pass for the Futhark compiler that compiles irregular nested data parallelism into flat parallel GPU code. Chapter 3 presented the transformation on a small source and target language: distribution rewrites a map nest into a sequence of single-statement maps (Section 3.3), the flattening rules replace these maps with segmented operations (Section 3.4), and the segmented operations are lowered to flat parallel primitives over a representation of irregular arrays as flat data together with segment metadata (Sections 3.5 and 3.6).

We then showed that many of the classical costs of full flattening can be avoided through several optimisations (Chapter 4). Most importantly, uniform statements can be recognised and handled without falling back to the general machinery for irregular computations. Scalar statements can be grouped together, which avoids distributing the map over each individual statement (Section 4.2). The replication of variant free variables is also left implicit through the `Replicated` representation, and replicated values are materialised only when an operation requires a dense representation (Section 4.1). Finally, because exploiting all available parallelism is not always profitable, when possible, the pass generates several semantically equivalent versions of each parallel nest in the style of incremental flattening and defers the choice among them to run time through tunable thresholds (Section 4.5). The transformation was further extended to conditionals, rearrange and functions in Chapter 5, while the implementation covers the full Futhark language, including constructs such as loops, scans, and histograms (Chapter 6).

The evaluation (Chapter 7) supports five claims. First, testing provides empirical evidence that the transformation preserves program semantics. The pass succeeds on all 2544 test cases in the Futhark test suite, on new tests written specifically to cover the irregular cases of each flattening rule, and on the programs in the Futhark benchmark suite (Section 7.1). Second, it improves expressiveness, as programs whose intermediate shapes are discovered only at run time, such as a growing loop-carried array inside a map, were previously rejected by the GPU backend but can now be compiled and executed (Section 7.2). Third, it mostly preserves the performance that Futhark already achieves on regular programs. On the Futhark benchmark suite, the geometric mean speedup over the existing compiler is 0.99, and only 9 of the 96 entry points regress by more than 10% (Section 7.3.1). Where regressions occur, they are

primarily caused by untuned thresholds, rather than by an inherent limitation of the approach. Fourth, it substantially improves the performance of irregular programs. On sparse-matrix–vector multiplication over synthetic workloads, the natural nested formulation is 296× faster on average than when compiled by the existing compiler, making irregular programs that previously exposed insufficient parallelism practical on the GPU (Section 7.3.2). Fifth, this comes at a modest cost in compile time. Across the benchmark suite, the pass increases compile time by 4.7% on average, with a median overhead of 1.2% (Section 7.4).

Taken together, these results show that full flattening of irregular nested parallelism can be integrated into a mature compiler without sacrificing its performance on regular parallelism. Programmers can now express irregular algorithms in their natural nested form while still obtaining efficient GPU code, where previously they often had to choose between manually flattening the program and abandoning the GPU backend.

8.1 Limitations and Future Work

The implementation has a number of limitations, each of which points to a concrete direction for future work. We state them together here.

Multi-version function lifting. The lifted-function convention is conservative in two respects, both because a single lifted version must serve every call site. First, every array parameter is assumed irregular, even when every call passes a regular array, which forces the construction of irregular metadata that a more precise scheme could avoid. Second, every parameter is assumed variant. Consider a function such as:

```
def f (b: bool) (x: []i32) =
  if b then e1 else e2
```

In the lifted function the boolean parameter is turned into an array, and the conditional is handled like a non-uniform conditional, because the parameter is now a variant array. But a caller might pass an invariant boolean, which does not need to be turned into an array. In that case the boolean could stay scalar and the conditional could be treated as a uniform one (Section 5.1.1), which would be much faster. Because almost all parallel functions in Futhark are currently inlined, this limitation had little effect in the benchmark suite (Section 6.8), but it would matter for programs that rely on non-inlined parallel functions. Generating multiple lifted versions, specialised by the regularity and variance of their parameters, and selecting among them at the call site, would address both problems. The trade-off is a potential explosion in the amount of generated code, so a practical scheme would have to limit which specialisations are produced.

Statement reordering during distribution. Vectorisation avoidance groups consecutive scalar statements, but the implementation never reorders statements to bring

together scalar statements that are separated by an independent parallel statement. Consider the following expression:

```
map (\x ->
       let z = x + 1
       let ys' = map (+x) ys
       let t = z + 100
       let r = t + ys'[0]
       in r)
    xs
```

With distribution as it currently works, the result is:

```
let zs   = map (\x -> x + 1) xs
let yss' = map (\x -> map (+x) ys) xs
let rs   = map (\z ys'' ->
                  let t = z + 100
                  in t + ys''[0])
               zs yss'
```

This is not the optimal grouping. The parallel statement sits in the middle of the scalar statements, but it does not depend on `z`, so `z = x + 1` could be moved past it and grouped with the later scalar statements:

```
let yss' = map (\x -> map (+x) ys) xs
let rs   = map (\x ys'' ->
                  let z = x + 1
                  let t = z + 100
                  in t + ys''[0])
               xs yss'
```

which uses one fewer kernel and one fewer intermediate array. Reordering statements in this way requires only a dependency analysis and could be implemented as a step in preprocessing.

Intra-block code generation in more cases. The intra-block version is currently not generated when the nest contains irregular parallelism, nor when the map body calls a parallel function (Section 6.8). Losing the intra-block version means losing shared-memory reuse, and this is the largest single source of regression observed in the evaluation. On the unmodified `lud` benchmark, which relies on intra-block execution, our pass is roughly 1000× slower than the existing pass, and the numbers reported in Section 7.3.1 recover only because the inner parallelism was explicitly annotated `#[sequential]`. A possible direction for future work is to make intra-block execution interact with selective sequentialisation. Instead of rejecting the intra-block version as soon as any irregular nested parallelism is encountered, the compiler could choose a cut-off point in the nest and sequentialise the parallelism below that point. In some cases, the compiler would then never encounter the irregular parallelism in the intra-block version, because the part of the nest that contains it has already been

sequentialised. This would allow the compiler to preserve shared-memory reuse in the surrounding block-level computation, even though some deeper parallelism has been sacrificed.

Operators with variant free variables. Reductions, scans, and histograms whose operators use a free variable that is variant in the surrounding nest are currently handled by the sequentialising fallback (Section 6.6). The reason is a limitation of the flat segmented operations of the GPU representation. Such operators are rare in practice, but supporting them would require extending the segmented operations of the GPU representation.

Bibliography

[1] G. E. Blelloch, "NESL: A nested data-parallel language (version 3.1)," Tech. Rep. CMU-CS-95-170, Carnegie Mellon University, 1995.

[2] G. E. Blelloch and G. W. Sabot, "Compiling collection-oriented languages onto massively parallel computers," *J. Parallel Distrib. Comput.*, vol. 8, p. 119–134, Feb. 1990.

[3] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.

[4] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pp. 556–571, ACM, 2017.

[5] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea, "Incremental flattening for nested data parallelism," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pp. 53–67, ACM, 2019.

[6] C. E. Oancea, *Lecture Notes for the Software Track of the PMPH Course*. University of Copenhagen, 2018.

[7] C. Sevald-Krause, "Flattening irregular nested parallelism in Futhark." BSc thesis, Department of Computer Science, University of Copenhagen, 2023.

[8] W.-m. W. Hwu, D. B. Kirk, and I. El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 4th ed., 2022.

[9] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[10] T. Henriksen, *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, 2017.

[11] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[12] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[13] T. Henriksen and M. Elsman, "Towards size-dependent types for array programming," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2021, pp. 1–14, ACM, 2021.

[14] Futhark Contributors, "Futhark: An optimising compiler for a functional, array-oriented language (compiler architecture)." Haddock documentation for the `futhark` package, version 0.25.33. Available: https://hackage-content.haskell.org/package/futhark-0.25.33/docs/Futhark.html. Accessed 2026-06-15.

[15] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pp. 237–247, ACM, 1993.

[16] L. Bergstrom and J. Reppy, "Nested data-parallelism on the GPU," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pp. 247–258, ACM, 2012.

[17] Y. Zhang and F. Mueller, "CuNesl: Compiling nested data-parallel languages for SIMT architectures," in *Proceedings of the 41st International Conference on Parallel Processing*, ICPP '12, pp. 340–349, IEEE, 2012.

[18] J. Reppy and N. Sandler, "Nessie: A NESL to CUDA compiler," in *Compilers for Parallel Computing (CPC '15)*, (London, UK), 2015.

[19] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data parallel Haskell: A status report," in *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pp. 10–18, ACM, 2007.

[20] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones, "Vectorisation avoidance," in *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pp. 37–48, ACM, 2012.

[21] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw, "Data-only flattening for nested data parallelism," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 81–92, ACM, 2013.

[22] M. Elsman, T. Henriksen, and N. G. W. Serup, "Data-parallel flattening by expansion," in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, pp. 14–24, ACM, 2019.

[23] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. Chakravarty, "Harnessing the multicores: Nested data parallelism in Haskell," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (2008)*, pp. 383–414, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2008.

[24] The Futhark Hackers, "`futhark-benchmarks`." https://github.com/diku-dk/futhark-benchmarks. Accessed 2026-07-08.

[25] D. Peter, "hyperfine." https://github.com/sharkdp/hyperfine, Mar. 2023. MIT License.