UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Efficient Sequentialization of Parallelism

Christian Marslev (wlz299) & Jonas Grønborg (lpg461)

Primary Advisor: Cosmin Eugen Oancea
Secondary Advisor: Troels Henriksen

February 2, 2024

Abstract

Writing parallel programs can be difficult, but can be offloaded by compilers through code transformations. Since parallelism is a finite resource on actual hardware, once the parallelism available is saturated, choosing to sequentialize some parts of computation can potentially result in speedups. In this thesis, it is shown how efficient sequentialization can be implemented as an automatic code transformation technique targeting intra-block parallelism on GPUs. Transformation rules and implementations are shown for the four parallel operators, map, reduce, scan, and scatter in the Futhark compiler. The optimization showcases how affected programs can achieve a speedup of 2 by introducing additional sequentialization to otherwise parallel programs.

Contents

1 Introduction

With the continuous rise of compute-intensive applications processing vast amounts of data and the increasing complexity of the underlying parallel hardware architectures, programming models that enable programmers to take advantage of this hardware are needed. GPGPUs are at the very core of this trend and through language extensions like CUDA and OpenCL, programmers can take advantage of the massive parallelism available on the hardware for accelerating their programs. However, optimizing programs in this manner often leads to expensive and unmaintainable programs, calling for another approach. One possibility is to offload the work to a compiler that can perform the optimizations automatically. Mapping a program that can express unlimited parallelism to parallel hardware with finite levels available is however challenging. Determining how to utilize the hardware requires smart compiler transformation techniques to do this efficiently.

One such technique involves the concept of *efficient sequentialization*. Sequentialization can be said to define the division of labor. A parallel program that computes n results may achieve this through n threads, each computing a single result. However, the same result can be achieved with n/q threads, if each thread produces q results. Here q is dubbed the *sequentialization factor*. In this way, sequentialization can be used as a code transformation technique to sequentialize excess parallelism.

In the CUDA programming model, parallelism can only be expressed in two levels on a GPU, at the grid and block level, and there is a finite cap to the number of threads that can execute in parallel. Once the parallelism of the hardware is saturated, there are no more benefits for increasing the degree of parallelism. Instead, it can be beneficial to sequentialize computations.

This thesis will explore how efficient sequentialization can be applied as an automatic compiler optimization for code running at the intra-block level on a GPU, and assess its potential with regard to the never-ending quest for performance. Specifically, the project will implement efficient sequentialization as a compiler pass inside of the compiler of the functional data-parallel programming language Futhark [19]. First by developing code transformation rules for four of the most commonly used parallel operators, **`map`**, **`reduce`**, **`scan`**, and **`scatter`**, and then showcasing how the language is extended to support the transformations. Finally, the optimization will be assessed in terms of the speedup achieved in several programs running with and without the optimization.

1.1 Terminology

Throughout this thesis, we will use terminology for concepts that sometimes have different names in different contexts. One such is the naming of the different levels of hardware parallelism. Futhark itself uses OpenCL naming convention with *group*, whereas CUDA uses *block*. We will use both interchangeably. Also, even though **`scatter`** is not a SOAC per definition, it will sometimes referred to as such since it is used in the same contexts. Thus, **`map`**, **`reduce`**, **`scan`**, and **`scatter`** may be referred to both as the SOACs or the parallel operators.

1.2 Repositories

All code for this project is available at the following repository links:

https://github.com/CKuke/futhark-seq/
https://github.com/kaff3/intraseq/

The most central parts are also included in the appendices B, C, D and E.

2 Related work

We were not able to find any other literature that specifically explores the concept of compiler-driven efficient sequentialization of parallelism. However, much has been written about the difficulty of maintaining high-performance programs across a variety of different platforms and the different approaches to achieving this. This is similar to how a platform-agnostic language like Futhark is designed to have a highly optimizing compiler generate efficient code for a variety of platforms through code transformations and an internal representation of parallel constructs while being completely invisible to the programmer. Thus, many of these ideas are related to the motivation for implementing an optimization like efficient sequentialization.

Ben-Nun et al. [2] introduce the concept of *performance-portable* which is used to characterize a code base that from the domain programmer's view of "what is being computed" remains unchanged as the code is optimized for different architectures. To achieve this, they present the idea of a data-centric intermediate representation enabling separating program definition from its optimization dubbed the *Stateful DataFlow multiGraph (SDFG)*. The motivation for SDFG is to enable the separation of the work of the domain programmers from the work of the performance programmers. This is because architecture-specific performance optimizations do not modify the computations of a program, but can still vary largely between platforms. One prime example of this is data movement since data locality is central to high-performance computing, but the implementation-specific details can vary largely between platforms. To preserve performance portability, Ben-Nun et al. therefore present a full implementation of a programming environment dubbed *DaCe* where the domain programmer can work in a well-known language like Python specifying the computations of the program. A compiler then transforms this into an SDFG which the performance programmer can work on to define the performance transformations.

Others have also explored the concept of performance portability. Moses et al. [13] highlight how parallelism is the main source of performance, but the diverse field of architectures in the parallel hardware landscape often leads to expensive application re-engineering. Language extensions like OpenCL have been proposed as a solution, but still require legacy applications to be reworked. The main contribution of the paper is a compiler-based solution to this problem. Specifically, they present a fully automated compiler based on a high-level and platform-agnostic representation of SIMT-style (single instruction, multiple threads) parallelism that can translate code from CUDA into a binary targeting CPU threads. They achieve an efficient translation partly due to a refined analysis of barrier semantics. Synchronization barriers are traditionally treated by compilers as forbidden territory regarding code transformations, but this is solved by extending the internal compiler representation of barriers to include memory properties including locations and access types.

One of the primary components of the efficient sequentialization transformations presented in this thesis is taking advantage of the different kinds of memory of the GPU architecture to achieve good performance. This is, however, not always straightforward as the fastest types of memory are typically limited resources. In their work on implementing automatic differentiation, Moses et al. [12] have explored the tradeoff of generating code that uses shared memory and registers on the GPU for fast access while keeping a balance such that it does not lower the available parallelism if the kernel requests too much.

3 Theory

This section will present the theory that this thesis builds upon. First, a high-level overview of the Futhark programming language will be provided by introducing the *SOACs* and the **`scatter`** operator central to Futhark programming and this thesis. A cost model will then be introduced that is used to argue the performance of these parallel operators based on an idealized hardware model called *PRAM*. Following this, a brief introduction to the concepts of *flattening* and *fusion* is given, both central to Futhark and to the actual implementation of the transformations inside of the Futhark compiler. Next, we will define what is meant by efficient sequentialization and show how *list homomorphisms* provides a mathematical foundation for this kind of optimization. Furthermore, some motivational context will be provided as to why sequentialization can result in performance gains.

3.1 Futhark

Futhark [10] is a statically typed, data-parallel, and purely functional array language designed to be compiled to efficient parallel code. Through a heavily optimizing compiler, data-parallel array computations are accelerated by utilizing the computing power of some underlying parallel hardware such as multi-threaded CPUs, or GPUs. The language supports regular nested parallelism as well as an imperative style of in-place updates of arrays.

Futhark programs are typically expressed in terms of a set of combinators that due to their parallel semantics enable the compiler to generate parallel code that performs bulk transformations of arrays. Most significant are the *second-order array combinators (SOACs)* that constitute the core of Futhark programming [19].

3.1.1 Second-Order Array Combinators

A SOAC is a second-order array combinator that transforms an array based on a functional argument. These can be arbitrarily nested and one of the primary goals of the Futhark compiler is then to map this nested parallelism efficiently to parallel hardware, as further explored in section 3.1.4. We will now present the types and semantics of each of the three basic SOACs, namely, **`map`**, **`reduce`**, and **`scan`**, together with simple examples of them in Futhark that could be mapped to intra-group parallelism by the Futhark compiler.

Map

The **map** SOAC is used to apply some function to all elements of some array, producing an array of the same size. but possibly of a different type. The types and semantics of the map SOAC can be seen below. The type signature states that **map** takes as its first argument a unary function that accepts an argument of type α and produces something of type β, and takes as its second argument, an array with elements of type α of length n. As the function is mapped to each value of the input array the return type is thus $[n]\beta$.

$$\begin{aligned} \textbf{map} &: (\alpha \to \beta) \to [n]\alpha \to [n]\beta \\ \textbf{map}\ f\ [a_1, \ldots, a_n] &= [f\ a_1, \ldots, f\ a_n] \end{aligned}$$

As an example, a map nest could be used to update each element of a matrix, in Futhark expressed as a two-dimensional array. Given an array `xss` of integers with dimensions $[n][m]$, the Futhark program in listing 1 would add 2 to each element of `xss`. The idea is that the outer map would, as an argument, get each inner array, `xs`, of size $[m]$ and then use an inner **map** to map over all m elements.

```
1 map (\xs ->
2     map (\x -> x + 2) xs
3 ) xss
```

Listing 1: **map** nest in Futhark

Reduce

The **reduce** SOAC iterates over all elements of some array, reducing it to a single result. The types and semantics of the **reduce** can be seen below. The **reduce** SOAC takes, as the first argument, some associative binary function, $\oplus$, some neutral element of said operator, $0_\oplus$, and finally an array of some type $[n]\alpha$. The result is then produced by applying the $\oplus$ operator to each element of the input array, for each element supplying the current element and the current intermediate reduction result as arguments to $\oplus$, resulting in a final single value of type α once all elements are processed.

$$\begin{aligned} \textbf{reduce} &: (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha \\ \textbf{reduce}\ \oplus\ 0_\oplus\ [a_1, \ldots, a_n] &= 0_\oplus \oplus a_1 \oplus \cdots \oplus a_n \end{aligned}$$

Similar to the previous **map** example, given an array of integers, `xss`, with dimensions $[n][m]$, a **reduce** nested inside of an outer **map** could be used to compute the sum of all n inner arrays of size m. This is shown below in listing 2.

```
1 map (\xs ->
2     reduce (+) 0 xs
3 ) xss
```

Listing 2: **reduce** nest in Futhark

Scan

The **scan** SOAC works very similarly to **reduce**, but instead of a single result, it produces an array of all the intermediate results. A common use of **scan** is to compute the prefix sum of some array. The types and semantics of **scan** can be seen below. As the first argument, it takes an associative binary operator, $\oplus$, some neutral element of said operator, $0_\oplus$, and lastly the array of type $[n]\alpha$ to perform the scan over. As such the return type is also of type $[n]\alpha$.

$$\textbf{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$
$$\textbf{scan}_{\ incl}\ \oplus\ 0_\oplus\ [a_1, \ldots, a_n] = [0_\oplus \oplus a_1,\ 0_\oplus \oplus a_1 \oplus a_2,\ \ldots,\ 0_\oplus \oplus a_1 \oplus \cdots \oplus a_n]$$

scan in Futhark is *inclusive* where the first element of the result array is the result of applying the operator to the first input of the input array and the neutral element, as shown in the semantics above. An alternative version is the *exclusive* **scan**, where the first element of the result array is simply the neutral element. The semantics of *exclusive* **scan** is shown below.

$$\textbf{scan}_{\ excl}\ \oplus\ 0_\oplus\ [a_1, \ldots, a_n] = [0_\oplus,\ 0_\oplus \oplus a_1,\ \ldots,\ 0_\oplus \oplus a_1 \oplus \cdots \oplus a_{n-1}]$$

An example of using **scan** to compute the prefix sum of each row of a matrix in Futhark is given below in listing 4. `xss` is a two-dimensional array of size $[n][m]$ so by nesting the **scan** inside of a **map**, each row of size m is scanned, producing a new array containing the prefix sum of all inner arrays.

```
1 map (\xs ->
2     scan (+) 0 xs
3 ) xss
```

Listing 3: **scan** nest in Futhark

3.1.2 In-place updates

Another core feature of Futhark is the notion of in-place updates. Traditionally, in a pure language, an array update would take time proportional to the size of the array as the entire array would have to be copied. This can, be mitigated, if it can be guaranteed that the original array will not referenced after the update has taken place. This way, the array can simply be updated in place effectively reducing the work required to be proportional to the single update. Futhark achieves this through its type system, allowing for bulk in-place updates of arrays in parallel. [10]

With in-place updates being a cornerstone of Futhark programming, these are also of great interest to this project since supplying transformation rules for **scatter** will significantly increase the set of programs to which the efficient sequentialization optimization can be applied.

Scatter

In Futhark, updates are commonly expressed through the use of the scatter function. The type signature of **`scatter`** can be seen below.

$$\textbf{scatter} : [m]\alpha \to [n]\texttt{i64} \to [n]\alpha \to [m]\alpha$$

The **`scatter`** function is used to perform bulk parallel in-place updates of an input array. The first argument of type $[m]\alpha$ is the array to be updated. The second array of type $[n]\texttt{i64}$ is the indices to write to in the array of the first argument and the final argument of type $[n]\alpha$ is the values to write to the array of the first argument. The most clear way of describing the semantics of scatter is with an imperative loop. Let `ds`, `is` and `vs` be the first, second, and third argument respectively, then the semantics can be described by the loop below.

```
for j = 0 to n do
    let i = is[j]
    let v = vs[j]
    ds[i] = v
```

In Futhark, **`scatter`** ignores out-of-bounds writes and since it is a parallel operator, any duplicate values in the array of indices to write to are considered undefined behavior [6].

A simple example of a Futhark program using **`scatter`** to update an array is given below where `dss` is the array to update, `iss` are the indices to update, and `vss` are the values to write to the indices. Note that Futhark only accepts this program if it can ensure that the original `dss` array would not be used afterward.

```
1  map3 (\ ds is vs ->
2    scatter ds is vs
3    ) dss iss vss
```

Listing 4: In-place updates with **`scatter`** in Futhark

The **`scatter`** function is not an actual SOAC as it takes no functions as arguments. We will however throughout the report reference all these 4 operators as *the SOACs* for simplicity.

3.1.3 Cost Analysis in Data Parallel Programming

Since Futhark is a data-parallel programming language, we need some way to reason about the complexity of parallel programs. For instance, for a sequential implementation, the complexity of a **`map`** over an array of size *n* would be *O(n)*, but a parallel implementation running on a machine with *n* available threads would only be *O(1)*, as each thread could process an element in parallel.

In this section, we will discuss the properties of parallel programs using an idealized hardware model named *parallel random access machine* (PRAM) and use it to argue the complexity of the parallel operators presented. This model ignores issues related to synchronization and communication and further assumes that [16]:

- There are P processors all with a unique identifier i such that $0 \leq i \leq P$.
- All processors are connected to the same shared memory.
- All processors execute instructions in lock-step, i.e. a processor cannot start the next instruction until all cores have finished the previous. If a branch is encountered, the processors not on the branch will have to wait for the ones that are to finish.
- Each parallel instruction takes unit time.

With this in mind, we define the work-depth asymptotic complexity of a parallel program running on a PRAM machine with $P = \infty$ (i.e. an infinite amount of processors) as [16];

- **Work complexity.** The total number of operations, across all processors, required to run the program. It is denoted as $W(n)$ where n is the size of the workload.
- **Depth complexity**. The amount of sequential steps needed to run the program. It is denoted as $D(n)$ where n is the size of the workload.
- **Work efficient**. A program is said to be work efficient if its work complexity is asymptotically equivalent to that of the best-known sequential algorithm.

Naturally, real machines do not have an infinite amount of processors, but knowing the work and depth of an implementation on a PRAM machine enables us to derive a bound on the complexity of the program on a PRAM machine with a finite P amount of processors. To do this, we use Brent's theorem as defined below:

Theorem 1 (Brent's Theorem [16])**.** *A parallel implementation that has Depth $D(n)$ and work $W(n)$ can be simulated on a P-processor PRAM in time complexity T such that;*

$$\frac{W(n)}{P} \leq T \leq \frac{W(n)}{P} + D(n)$$

With these definitions, we can analyze the work and depth complexity of all the operators of this paper. The results are summarised in table 1. The actual work and depth complexity of the three SOACs depends on the complexity of either the function f used in **`map`** or the operator $\oplus$ used in **`scan`** and **`reduce`**. For simplicity of this analysis, we assume the work and depth of both are constant. For some input a of size n we have;

- **Map**. As the function f has to be applied once to every element of a, it must be applied n times giving a work complexity of $W(n)$. As the model assumes an infinite amount of processors all these applications can be done in parallel giving a depth complexity of $D(1)$.
- **Reduce**. The reduction of an array of length n can be achieved by giving two elements to each processor that would apply $\oplus$, meaning we would need $\frac{n}{2}$ processors. After the first step, there would then be $\frac{n}{2}$ elements to be processed and so on, leaving $\frac{n}{2^i}$ elements at step i. This would give a work efficient complexity of $W(n)$. For the depth, assume $n = 2^k$ for some $k \geq 1$. We have to look at how many of these reduction steps are needed. At step k, we will have the remainder of the elements being $\frac{n}{2^k} = 1$. Since $n = 2^k$ then $k = \lg 2^k = \lg n$, meaning we have depth complexity of $D(\lg n)$.
- **Scan**. The algorithm for parallel **`scan`** is slightly more involved and we will not delve into the specifics here. However, **`scan`** can be implemented as first a parallel **`reduce`** and then adding one more pass of the same structure with the same work and depth. Thus, the same reasoning for used **`reduce`** can be applied to **`scan`** since they only diverge in the constant factors, giving $W(n)$ and $D(\lg n)$. [16]
- **Scatter**. This analysis is straightforward. We have n elements split between n processors, giving $W(n)$ and as they can all be done in parallel we have $D(1)$.

The work and depth complexities of the parallel operators introduced are summarised below in table 1.

	Work	**Depth**
`map`	$W(n)$	$D(1)$
`reduce`	$W(n)$	$D(\lg n)$
`scan`	$W(n)$	$D(\lg n)$
`scatter`	$W(n)$	$D(1)$

Table 1: Work and depth complexity of the SOACs

3.1.4 Flattening

We have now presented the basic parallel building blocks for writing parallel programs in Futhark. The next logical step then is how the Futhark compiler generates code for these programs. As shown, SOACs can be arbitrarily nested, but the hardware has a finite amount of parallelism to exploit, so the question is how should these programs be mapped to physical hardware. One of the main goals of the Futhark compiler is how to perform this mapping and it is necessary to have a basic understanding of this process in order to make any changes to the code generated by the compiler.

There have been many attempts at code transformations for improving the amount of nested parallelism that is mapped to hardware, but common for them all is that the program produced is only truly efficient for one class of workloads [11]. One such transformation that has been broadly explored [4] [10] [11] is the concept of *flattening*. Flattening is a transformation that rewrites a program exhibiting nested parallelism into a semantically equivalent one operating on one-dimensional vectors.

However, Futhark does not rely on a "one-size-fits-all" approach by choosing a specific mapping of par-

allelism to hardware and producing a single code version [11]. Instead, the compiler utilizes *incremental flattening* which is a compiler-driven analysis that clusters datasets by several near-optimal code versions that are discriminated at runtime. The compiler does this by introducing different code versions, guarded by some threshold, at each point where it encounters a **`map`** operator that is to be mapped to some level l of hardware parallelism. Essentially, the thresholds determine what degrees of sequentialization and flattening there should be applied at runtime. [11]. After applying incremental flattening the different code versions will end up in a tree-like structure as illustrated in figure 1. Here, $V1, \ldots, V4$ illustrate different code versions generated, (n_1, n_2, n_3) is some dataset and (p_1, p_2, p_3) are some threshold parameters used to determine which code version will be run.

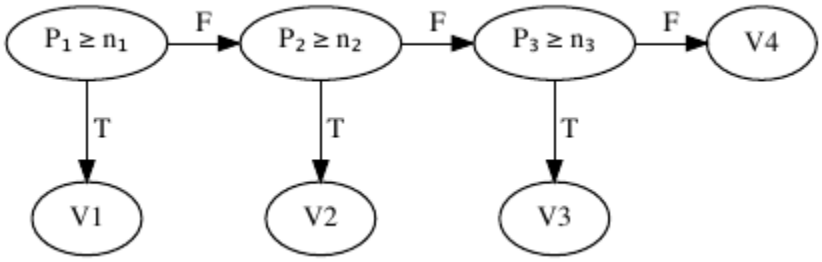


Figure 1: Branching tree produced by incremental flattening [11].

This distinction between different code versions is particularly important to this thesis, as the optimizations specifically target the intra-group code version of the generated GPU code. This is a code version in which some outer **`map`** has been mapped to hardware level 1, i.e. the grid, and remaining SOACs to hardware level 0, which is equivalent to a block. These different levels will be discussed further in section 3.2. The optimization will be applied inside the compiler at a stage where the different versions have already been generated, so it is important to operate on the correct code version. Working at the intra-group level also means we can assume that locally created arrays will be allocated in shared memory [11]

Once our implementation has been applied, it will have changed the number of threads used by the intra-group SOAC, meaning the threshold for determining if the intra-group version should run, should also be modified. This is however outside the scope of this thesis, and as we shall see later a program can be annotated to specify that only the intra-group code versions should be produced.

3.1.5 Fusion

Another core part of the Futhark compiler internals is the concept of *fusion*. Fusion is a code transformation technique that combines multiple code constructs into one. For instance, using a **`map`** to transform an array with the function f and then transform the result with another **`map`** with the function g is effectively the same as transforming the array with a single (fused) **`map`** with the function $g \circ f$ [9]. Fusion is widely used in Futhark as a way to remove the need for intermediate arrays and iterating more times than necessary over the same array, while still allowing the programmer to express their programs as simple compositions of SOACs.

While fusion does not have any direct impact on the transformations presented in this thesis, the concept is largely reflected in the compiler internals where several code constructs like `SegOp`s and `Screma`s, described in section 6.2, are designed to handle multiple fused SOACs.

3.2 GPU Programming model

While this project will not be making any direct changes to the GPU code itself generated by the compiler, it is still necessary to have a basic understanding of GPU programming to achieve desirable performance. Different programming models exist for GPUs including CUDA and OpenCL which both provide an abstraction level between the program and the actual hardware implementation of the specific GPU. In this section, we will use the CUDA programming model [14] to introduce these different concepts. This will only be a brief overview with more information available at [14].

The CUDA programming model assumes that CUDA threads execute on a discrete *device* that acts as a coprocessor to the *host*. This is the case when the program mainly runs on the CPU and launches kernels to run on the GPU. It is assumed that the host and device have separate memory spaces called *host memory* and *device memory* with functionality for copying data between both.

Thread Layout

In CUDA, threads are organized into two parallel levels, namely, *thread blocks* which themselves are grouped into a *grid* of multiple thread blocks as illustrated in figure 2. The maximum number of threads pr. block is 1024. This means if, for some problem, each thread will process a single element, and if the problem size $n > 1024$, then multiple blocks are needed. However, all blocks within a grid have the same size, even if not all threads for some of the blocks are going to be used. CUDA exposes a set of variables that can be used to identify the given thread and block inside of a kernel. The two most noteworthy are `blockIdx.x` and `threadIdx.x` which can be used to get the ID of the current thread and in which block it is executing. This is usually used for determining which elements a specific thread should access.

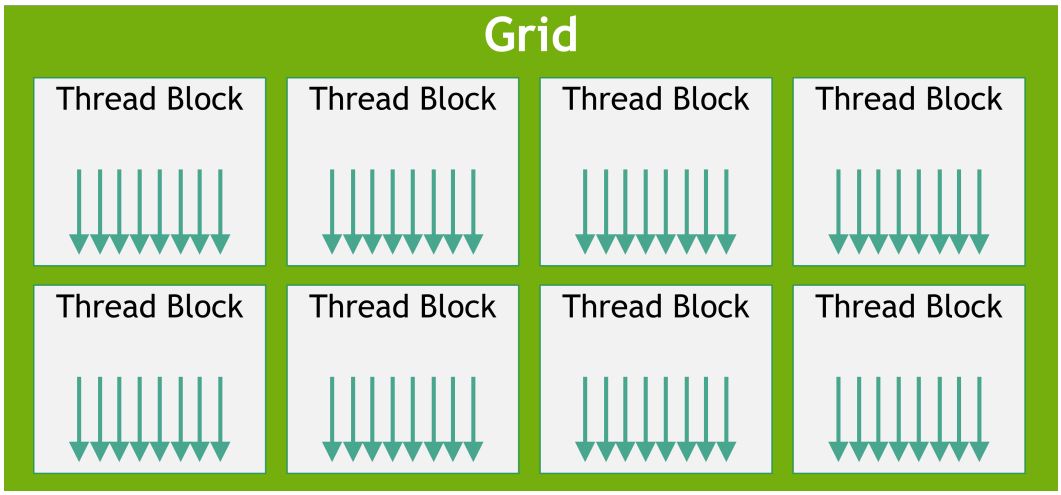


Figure 2: Grid with thread blocks [14].

Memory Hierarchy

Each thread may access data from multiple places during execution. All threads in a single block have some *shared memory*. That is, threads from across different blocks are not able to access the shared memory of the block the other thread resides in. This essentially works as a user-managed cache. The shared memory is split up into *banks*, the exact size of which is determined by the specific GPU *compute capability*. If n memory accesses across n threads are performed and these addresses fall into n distinct banks, they can all be served in parallel. If however multiple threads try to access the same banks a *bank conflict* occurs. When this happens the memory transfers are split up into as many conflict-free memory accesses as needed which will be served sequentially, reducing the overall throughput. Finally, there is *global memory* which is shared across all blocks in the grid and it is in this memory data is staged when copied from the host to the device. Since shared memory essentially works like an L1 cache and is expected to be near the processor core with low latency, it is much faster to access shared memory in comparison to global memory [14].

SIMT Execution

The hardware itself is built around an array of *streaming multiprocessors* (SMs) to which blocks of some kernel are distributed when launched by the host program. One SM can execute multiple blocks as long as they do not exceed the SM execution capacity. As blocks terminate, new blocks are launched to take up the vacated SMs.

The SM executes threads in groups of 32 parallel threads called *warps*. All such threads within a warp start at the same program address but are free to branch and execute independently. However, all threads within a warp execute in *lockstep* meaning that one common instruction is executed at a time, implying divergent threads within a warp will not execute at the same time, they will be *inactive* when the current instruction is not on their code path and *active* when it is. While all threads will at some point reach the same instruction again, it is possible to make sure all threads wait at specific points through synchronization and barriers.

This is especially useful when the program needs to be sure that all memory for some computation has been updated by all warps.

Performance Considerations

To achieve the best performance we want to minimize data transfers with low bandwidth by maximizing the use of on-chip memory in the form of caches and shared memory. One way to achieve this is to stage the data the kernel needs in shared memory, and perform the needed work there before finally writing the result back to global memory. If the block would need to access the data from global memory multiple times, we can instead do that only twice. Once for reading and once for writing.

We also want to consider the data access patterns when reading from global memory into shared memory as this will affect the throughput of memory by orders of magnitude [14]. An instruction that reads addressable memory might need to be re-issued multiple times depending on the memory addresses read by each thread across a warp. When a warp performs a memory access, all the accesses of the individual threads will be coalesced to one or more memory transactions. The amount of transactions depends on the size of the word accessed and the distribution of memory addresses used across the threads. As such we want to maximize the amount of coalesced access to ideally only have single memory transaction pr. warp by having the threads of a warp read consecutive memory addresses.

3.3 Efficient Sequentialization

Efficient sequentialization is a code transformation that sequentializes excess parallelism. Doing so allows the program to better utilize the hardware by reducing the amount of inter-thread communication. The transformation rests on a mathematical foundation based on *list homomorphisms*. This section will first introduce the mathematical background and use it together with Brent's theorem to show a better bound on the runtime of reduce on a P processor machine. The section will then end with a discussion of the runtime benefits the transformations give us on actual hardware.

3.3.1 List Homomorphisms

Homomorphisms are functions that match the divide-and-conquer paradigm and are thus well suited for parallel execution [7]. More formally we have;

Definition 1 ([3, 7]). *A function h defined on lists is a homomorphism if and only if there exists a binary associative operator $\oplus$ with neutral element $0_\oplus$ such that for all lists x and y we have*

$$h(x +\!\!+ y) = h(x) \oplus h(y)$$

where $+\!\!+$ is list concatenation and $h([]) = 0_\oplus$

The intuition here is that we can take any list $z = x +\!\!+ y$ where x and y are an arbitrary split of z and apply h to each sublist in isolation and apply $\oplus$ to combine the results. That is, we have a divide-and-conquer scheme in which the problem is split into two new subproblems that add up to the result of the original problem in the end. Further, if we define $fa = h([a])$ then h can be determined by f and $\oplus$ alone [3]. This leads us to the first homomorphism lemma.

Theorem 2 (First Homomorphism Theorem [3, 7]). *A function h is a homomorphism with respect to $+\!\!+$ if and only if it can be factored into the composition*

$$h = (\textbf{\textit{reduce}}\ \oplus\ 0_\oplus) \circ (\textbf{\textit{map}}\ f)$$

for some function f and binary operator $\oplus$. ***map*** *and* ***reduce*** *has the semantics as described in section 3.1*

From this theorem we can see that **map** and **reduce** by themselves are homomorphisms by the following [3]:

$$\textbf{map}\ f = (\textbf{reduce}\ +\!\!+\ []) \circ (\textbf{map}\ f)$$
$$\textbf{reduce}\ \oplus\ 0_\oplus = (\textbf{reduce}\ \oplus\ 0_\oplus) \circ (\textbf{map}\ id)$$

where id is the identity function.

Next up we have the *promotion lemmas* that can be seen as rewrite rules that allow us to optimize the program in various ways.

Lemma 1 (Promotion Lemmas [3, 16]). *Given unary functions f and g, an associative binary operator $\oplus$ with neutral element $0_\oplus$ then the following identities hold.*

1. $(\textbf{map}\ f) \circ (\textbf{map}\ g) \equiv \textbf{map}\ (f \circ g)$
2. $(\textbf{map}\ f) \circ (\textbf{reduce}\ (+\!\!+)\ [\,]) \equiv (\textbf{reduce}\ (+\!\!+)\ [\,]) \circ (\textbf{map}\ (\textbf{map}\ f))$
3. $(\textbf{reduce}\ \oplus\ 0_\oplus) \circ (\textbf{reduce}\ (+\!\!+)\ []) \equiv (\textbf{reduce}\ \oplus\ 0_\oplus) \circ (\textbf{map}\ (\textbf{reduce}\ \oplus\ 0_\oplus))$

1. The first identity is the same as the **map**-**map** fusion as described earlier. It essentially allows the program to reduce the amount of memory access. Instead of storing the result of **map** g first before applying **map** f the composition $f \circ g$ can just be used directly.
2. This rule is especially useful in the rightwards ($\Rightarrow$) direction. It states that if some function f is to be applied to all elements of some $[n][m]$ array flattened, then that is the same as applying f to all n inner arrays and concatenating the result. That is, if we have excess parallelism that the hardware cannot handle, then it is possible to sequentialize this [16]. That is assuming the innermost **map** would run sequentially.
3. Like for the second rule, this is useful in the rightwards ($\Rightarrow$) direction. Like before it states that we can sequentialize the excess parallelism, and do some of the reduction sequentially on multiple processors, before doing the final reduction across all the pr. processor results.

With these rules in hand, we can prove the *optimized map-reduce lemma*.

Lemma 2 (Optimized map-reduce [16]). *Assume an associative binary operator $\oplus$ with neutral element $0_\oplus$ and a function f, then the following always hold*

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

where $\textbf{split}_p :: [\alpha] \to [[\alpha]]$ *splits a list into* p *subsists all of the same size, and where* **redomap** *is defined as* $\textbf{redomap} \oplus f\ 0_\oplus \equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ f)$.

Proof. To prove the above equivalence we first note that $(\textbf{reduce}\ (+\!\!+)\ [\,]) \circ \textbf{split}_p \equiv \textbf{id}$, where **id** is the identity function. The intuition here is that if we split some array up in p chunks and simply concatenate them all again, we will have the original array as the output of the composed function. Using this we get the following derivation;

$$
\begin{aligned}
\textbf{redomap} \oplus f\ 0_\oplus &\equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ f) \\
&\equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ f) \circ (\textbf{reduce}\ (+\!\!+)\ [\,]) \circ \textbf{split}_p \\
&\equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{reduce}\ (+\!\!+)\ [\,]) \circ (\textbf{map}\ (\textbf{map}\ f)) \circ \textbf{split}_p \\
&\equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ (\textbf{reduce} \oplus 0_\oplus)) \circ (\textbf{map}\ (\textbf{map}\ f)) \circ \textbf{split}_p \\
&\equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ ((\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ f))) \circ \textbf{split}_p \\
&\equiv (\textbf{reduce} \oplus 0_\oplus) \circ (\textbf{map}\ (\textbf{redomap} \oplus f\ 0_\oplus)) \circ \textbf{split}_p
\end{aligned}
$$

where first the definition of **redomap** is applied, which is composed of the identity function. Then the promotion lemmas, in order 2, 3, and 1, are applied, when then finally the definition of **redomap** can be applied in the other direction. □

The results of this theorem tell us if we have some function f that is to be applied to all elements of some array, which is then to be reduced with $\oplus$, then that is the same as first splitting the array up into P equal sized chunks, reducing over each chunk to produce P results and then finally reducing over the P results to produce the single final reduction result.

This has great implications for efficient sequentialization of reductions. Essentially, for a **reduce** the proof enables us to partition it into two steps by first assigning each thread a chunk of elements that can be reduced sequentially before then doing a block-wide reduction over the sequential results. Given P chunks that split a total workload of size n, the optimized map-reduce theorem is essentially limiting the number of processors used to P. This means, that the depth of the parallel reduction is reduced which should result in speedup. Assuming a PRAM machine, we can apply Brent's theorem to show a new bound for this reduction, utilizing that $P \leq n$

$$
O\left(\frac{n}{P}\right) \leq O\left(\frac{n}{P} + \lg P\right) \leq O\left(\frac{n}{P} + \lg n\right)
$$

3.3.2 Practical Details

We have now seen that splitting up the work between multiple processors is mathematically sound. In our case, these *processors* are the threads within a thread block, which means this new *chunk* of work has to be performed sequentially as we have no more parallelism to exploit at that hardware level.

There are also additional benefits of doing such transformations that are not explicitly clear from the above lemmas and theorems, but more stems from how it would be executed in actual hardware. As briefly mentioned in the explanation of the first transformation in lemma 1, it saves on the number of memory

access needed to do the computation. If said memory to access were to global memory then savings would be far greater than if it were shared memory. The second and especially third transformations would allow for the work distribution between threads to be more balanced. If we were to just do a single reduction across the entirety of the array, we would have many threads idling as the computation is going down the reduction tree. Instead, it is preferable to have all the threads perform more initial work, each producing an intermediate result, and then have them reduce across these cooperatively. Then the same amount of work could also be achieved using fewer threads, theoretically letting more blocks execute in parallel, or just use the same amount of threads to process even more. There would still be threads idling, but not as many pr. block, and not for so long as before.

4 Preliminary Work

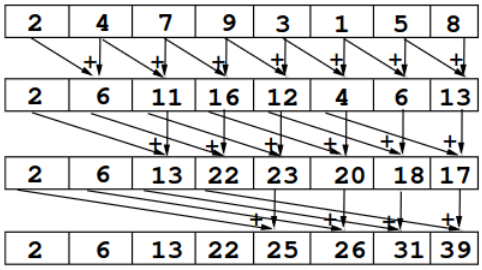
Before establishing the SOAC transformation rules, some preliminary work was done to investigate the effects of increased sequentialization. Specifically, an experiment with two different intra-block scan implementations using CUDA was conducted to give a rough estimation of the potential speedups. The effect of increasing the sequentialization factor was further investigated using an intra-block implementation of radix sort in CUDA.

4.1 Scan

To get a rough estimation of the potential speedups that can be achieved with sequentialization, two different versions of an intra-block inclusive **`scan`** were implemented in CUDA. **`scan`** was chosen as the operator since its transformation is the most complex, requiring several intermediate steps.

The first version is fully parallel where a thread is spawned for each element in the block to be scanned. The scan uses an intra-warp scan subroutine, see 3, to first do an inclusive scan of each warp. Each warp then copies its last element to the first warp which is then scanned again. Finally, the scanned results from the first warp can be used to accumulate the correct results across all warps. It should be mentioned that this algorithm is not work efficient as it has $W(n \lg n)$. However, since threads execute in lockstep on the GPU, this is the typical CUDA implementation for scanning a warp [16].

The second version is sequentialized where each thread processes $chunk$ number of elements by sequentially scanning its $chunk$ elements locally inside of its registers. The final element of each local scan is then written to shared memory which can then be scanned (using the first version as a subroutine) to get the offsets for each element. Finally, each thread can then update its $chunk$ elements sequentially and write back the results. This algorithm is outlined below in figure 4.




```
Input:  array A of n=2^k elements
                        of type α
        ⊕ : (α,α) → α associative
Output: B = [a_1, a_1⊕a_2, ... ,⊕_{j=0}^{n-1} a_j]
1.  forall i = 0 : n-1 do
2.    B[i] ← A[i]
3.  endfor
4.  for d = 0 to k-1 do
5.    h = 2^d
6.    forall i = h to n-1 do
7.      B[i] ← B[i-h] ⊕ B[i]
8.    endfor
9.  endfor
```

Figure 3: Warp-level inclusive scan [17]

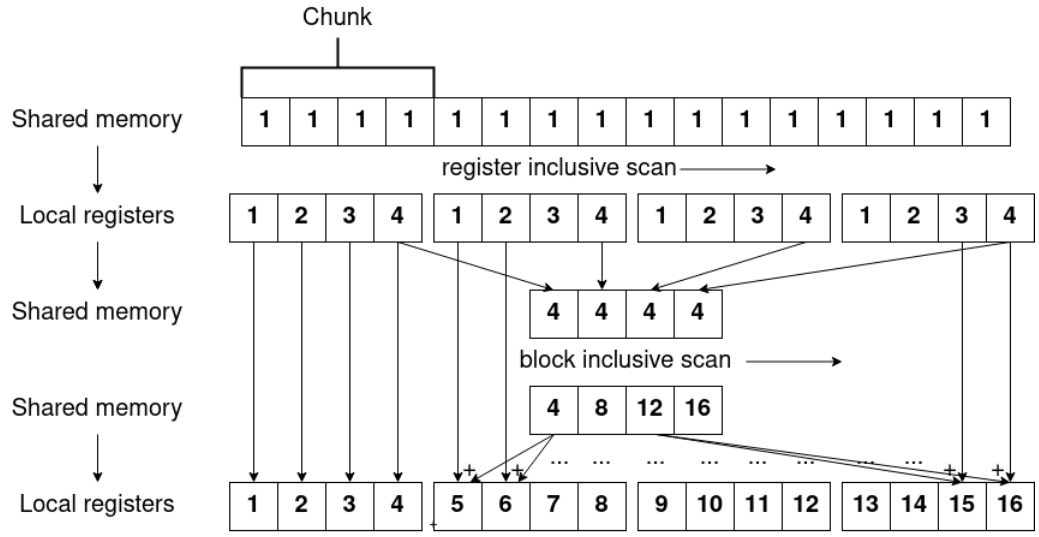


Figure 4: Sequentialized intra-block inclusive scan with `(+)` as operator. Each thread processes `chunk` elements

The results of comparing the two versions on different numbers of blocks can be seen in table 2. The results clearly show the potential of applying sequentialization with a consistent speedup of ≈ 1.8 and give an estimation of what can be expected for **`scan`**.

Number of blocks	100k	200k	300k	400k	500k
Speedup	1.89	1.83	1.82	1.81	1.81

Table 2: The speedup of the sequentialized scan version vs. the fully parallel on different number of blocks. The block size is fixed at 1024. Benchmarks run on NVIDIA A100.

4.2 Radix Sort

Radix sort is one of the oldest and best-known sorting algorithms. It assumes keys are d-digit numbers and sorts one key at a time. The complexity is $O(nd)$, but since d is considered constant, i.e. is decided upon beforehand and does not depend on the input the complexity is linear in n. The algorithm used for sorting within each of these d passes is usually counting sort, which is fairly easy to parallelize as a **`scan`** operation [18]. As such it has the potential to reap the benefits of efficient sequentialization.

Overall the algorithm can be split into three kernels [18]

1. Each block loads and sorts its tile in on-chip memory, computes a histogram with a bin for each possible digit, describing how many of the elements in the tile have the specific digit, and writes both back to global memory.
2. Performs a scan across all the block histograms to compute the global ranks of all elements within each tile.
3. Using the results of the previous scan, each block can now scatter their elements to the correct output position.

As we are working with the optimization of intra-group kernels, we will focus on the first of these kernels, each performing a block local radix sort.

To test the effects of efficient sequentialization on this kernel it has been implemented in CUDA in such a way that the sequentialization factor and group size can be altered through template parameters. Setting the sequentialization factor to 1 will therefore result in a fully parallel version, but since the implementation expects that each thread will process multiple elements, this does result in each thread doing some redundant work. However, this experiment is merely to show that sequentialization does impact the performance.

We wanted to test the effect of the sequentialization factor. We tested the implementation with a fixed block size of 256 threads, working on a data set of size 2^{27}. The results can be seen below in table 3. It is clear to see that having each thread process multiple elements increases the overall performance of the algorithm. While it shows that a factor of 23 is the fastest, the actual results of all, but that of a factor of 1, kept fluctuating. As such the only conclusion we can draw from is, is that efficient sequentialization is desirable, but not by what factor. In [18] from which our implementation is based, they use 4 elements pr. thread and so will we. Finally, the factor of 23 might seem odd but was found by NVIDIA to yield the best performance in their implementation of radix sort [1].

Sequentialization Factor	1	4	8	23
Run time (μs)	26829	14013	16008	13744

Table 3: Run time of running or implementation on a size of 2^{27}, with 256 threads pr. block and differing sequentialization factors. Run on NVIDIA A100.

5 Transformation Rules

This section will present the derived transformation rules for efficient sequentialization of the SOACs. Before introducing the transformation rules some notation will be defined that allows us to view the transformation through different lenses. First, a functional Futhark-like language is used as this gives the most intuitive idea of what the transformation is to achieve at a high-level. Second, a language similar to that of the *intermediate representation* used within the compiler where *segmented operators* are now the main parallel constructs will be used. As this is most similar to the representation the transformation is going to be applied to within the compiler it can be used to guide the implementation. Finally, these intermediate rules will also be expressed more imperatively with parallel loops and more explicit memory management for clarity's sake.

Originally formal rules were developed to be in line with already existing notation, like the one used for describing Futharks flattening algorithm [11]. However, these turned out to be more complex than needed for the transformations of this thesis. That said, they are still to be found in appendix A for the curious reader.

5.1 Notation

Neither of the languages used for the transformations needs a completely formal definition. The high-level language is essentially Futhark with some added benefits that allow us to focus on the problem at hand and not get bogged down in language specifics. Likewise, the intermediate language is much like the intermediate representation within the compiler but with a much simpler syntax similar to that of the high-level language. But as this language is a bit more esoteric still, more time will be spent on it, with imperative examples.

5.1.1 High Level Language

The high-level language is essentially Futhark. Variables are bound in let-statements and the parallel operators are the SOACs with the same semantics as described in section 3.1. To explain the efficient sequentialization transformation we introduce the $\mathbf{split}_p$ function, which given some array splits it up into p chunks of equal size. The type is;

$$\mathbf{split}_p \ : \ \texttt{[n]}\alpha \ \rightarrow \ \texttt{[n/p][p]}\alpha$$

5.1.2 Intermediate Language

The intermediate language has all the same syntax as the high-level one, but with the modified semantics in that, all the SOACs are now considered sequential. Further, we also allow the use of **redomap** and **scanomap** as a composition of **map**-**reduce** and **map**-**scan** composition respectively. They are defined as

$$\begin{aligned}
&\mathbf{redomap} \ \oplus \ \texttt{f} \ 0_\oplus \ \equiv \ (\mathbf{reduce} \ \oplus \ 0_\oplus) \ \circ \ (\mathbf{map} \ \texttt{f}) \\
&\mathbf{scanomap} \ \oplus \ \texttt{f} \ 0_\oplus \ \equiv \ (\mathbf{scan} \ \oplus \ 0_\oplus) \ \circ \ (\mathbf{map} \ \texttt{f})
\end{aligned}$$

Additionally, three new *segmented* operators **segmap**, **segred** and **segscan** are introduced as the parallel constructs. Formally they are to be considered a perfect map-nest on top of some bottommost computation of a **map**, **redomap** or **scanomap** respectively.

As we are working in a setting where there will always be two levels of parallelism, we will express these segmented operators in style with the actual intermediate representation. This is also more in line with how you would write an actual kernel in e.g. CUDA as launch parameters for the number of blocks and block sizes will be specified. Note, that the Futhark compiler conventionally uses OpenCL naming conventions where instead of *block*, *group* is used. Thus, to denote a segmented operator operating at the group(block) level, we will write

```
segmap(group, n, m):
    let x = ...
    in f x
```

where `n` specifies the number of groups and `m` the number of threads pr. group. The mapping part consists of reading and computing some value which is finally returned. However, before returning the map function `f` has to be applied on the value. All segmented operators nested inside will implicitly have `m` threads. We also use the magic variables `gid` and `tid` to denote the group id and thread id of the current group and thread respectively. This is akin to the `blockIdx.x` and `threadIdx.x` in CUDA.

As we are working on the intra-group version the outermost segmented operator, that is at group level, will always be a **segmap**. As such **segred** and **segscan** can only be defined on `thread` level. Like **redomap** and **scanomap** their segmented counterparts also require some binary operator $\oplus$ and neutral elements $0_\oplus$. For a segmented reduction, we would e.g. write;

```
segmap(group, n, m):
    segred(thread) ⊕ 0⊕:
        let x = ...
        in f x
```

in which each thread cooperating on the segmented reduce will read and compute some value `x` which is returned for the inter-thread reduction for the final result. However, note that before returning `x` some function `f` is applied to it. This is because a **segred** is in the end is a **redomap** which has a map function `f`. If `f` is simply the identity function, this would result in an inter-thread reduction of their respective elements.

Values can be read with the *slice* syntax, in which the total number of elements in each dimensions to be read has to be specified. For some array `xss` of shape `[n][m]`, `xss[i][j]`, would simply return a scalar, while `xss[i][0:m:1]` would return the entirety of the i'th inner array. Note that in the slice syntax `x:y:z`, `x` is the starting position of the slice, `y` is the number of elements to be read, and `z` is the stride to use when reading those elements.

Imperative

To solidify the semantics of nested segmented operators we will use a simplified, imperative-like language

in which the segmented operators have been replaced with parallel **forall** loops. That is, the loops can be understood sequentially, but they are executed in parallel. As an example the above **segmap**-**segred** nest can be described in terms of parallel loops as

```
forall gid < n do
   let ac = 0⊕
   forall tid < m :
      let x = ...
      ac = ac ⊕ (f x)
```

Furthermore, the imperative rules have been extended with a function `barrier()` that indicates where thread synchronization would be required. `scratch n` creates an array of size `n`. Finally, the following three functions **map** , **redomap** , **scanomap** are defined as following imperatively:

```
redomap ⊕ f 0⊕ xs
≡
let ac = 0⊕
for i = 0 < length xs do
   ac = ac ⊕ (f xs[i])
```

Listing 5: Imperative definition of **redomap**.

```
scanomap ⊕ f 0⊕ xs
≡
let tmp = scratch (length xs)
let ac = 0⊕
for i = 0 < length xs do
   ac = ac ⊕ (f xs[i])
   tmp[i] = ac
```

Listing 6: Imperative definition of **scanomap**.

```
map f xs
≡
let tmp = scratch (length xs)
for i = 0 to length xs do
   tmp[i] = f xs[i]
```

Listing 7: Imperative definition of **map**.

5.2 Rules

This section will introduce the transformation rules for each SOAC. They will be presented in both the high-level functional language and the one similar to the intermediate representation used inside of the compiler. Following these, we will also have the intermediate representation rules expressed as parallel loops with explicit barriers, to further solidify the understanding of the transformation. Having the rules at a high level gives the best intuition, coming from Futhark, on what the transformation is doing, while the intermediate rules are closer to the format that we are going to be working with within the compiler. They are however still high level enough that we do not need explicit memory allocation and can thus focus on the transformations themselves.

5.2.1 Map

```
Input: array xss[n][m]
Output: array of size [n][m]
---
map (λxs → map f xs) xss
≡
map (λxs →
  map (λx →
    map f x
  ) (split_{m/q} xs)
) xss
```

Listing 8: High-level map transformation rule.

```
Input: array xss[n][m]
Output: array of size [n][m]
---
segmap(group, n, m):
   segmap(thread):
      let x = xss[gid, tid]
      in f x
≡
let m' = ⌈m/q⌉
segmap(group, n, m'):
   let tile = xss[gid, 0:m:1]
   segmap(thread):
      let idx = tid·q
      let chunk = tile[idx:q:1]
      in map f chunk
```

Listing 9: Intermediate map transformation rule. `f` is the mapping function of the **segmap** in the original program.

The transformation rule for **map** in both formats can be seen in listings 8 and 9. It is assumed the array `xss` has shape `[n][m]`. This is the most simple of all the transformations as we do not require any intermediate steps. All that is happening is that the mapped over array `xs` is split into chunks on which the function `f` is then mapped. The transformation in the intermediate rules sheds some more light on how the data will be read and which parts are parallel and which are sequential. There, it is also possible to see explicitly that the number of threads has been reduced to `m'`. The final thing worth noting is that the splitting up of `xs` happens implicitly as each thread simply reads the elements needed directly from the tile which just comes from the `xss` array. A rule showing the imperative semantics is shown in listing 10.

Work

Assuming the work complexity of the map function `f` is constant, we have a work complexity of $W(\texttt{nm})$ as each inner **map** maps over a size `m` and that is done `n` times. In the transformed program the inner-most `map` works on `x` of size `q`, which is done `m'` times, which is all done `n` times, giving a work complexity of $W(\texttt{nm})$ as well.

It is worth noting why the reading of `tile` and the sequential **map** in the transformed program of the intermediate rule do not add to the work complexity. For the former, we are only reading `m` elements which is also linear. For the latter, it is simply because sequential work does not add to the work complexity. We will look at this again for the depth.

Depth

Assuming the depth complexity of the map function `f` is constant, we have a depth of $D(1)$. All `nm` elements can be processed in parallel so no sequential steps are needed. Now, recall that in the high-level

rules, we assume all SOACs are parallel meaning the depth of the transformed program is also $D(1)$. We can however see from the intermediate rule that not all parts are parallel. First, the reading of `tile` of size `m` happens using only `m'` threads. As such some of those elements must be read sequentially, but at most `q` elements pr. thread. However, as `q` is considered constant for any program, i.e. it does not depend on the input size, the reading of `tile` is constant depth. The same logic applies to the sequential **`map`**, which maps over an array of size `q`, but as `q` is constant this is also constant depth. As such, the transformation preserves the depth complexity of the program.

Imperative `map` rule

```
Input: array xss[n][m]
Output: array out[n][m]
---
forall gid = 0 to n do
   forall tid = 0 to m do
      let x = xss[gid, tid]
      out[gid, tid] = f x
≡
let m' = ⌈m/q⌉
forall gid = 0 to n do
   let tile = xss[gid, 0:m:1]
   barrier()
   forall tid = 0 to m' do
      let idx = tid·q
      let chunk = tile[idx:q:1]
      tile[idx:q:1] = map f chunk
   barrier()
   out[gid, 0:m:1] = tile
```

Listing 10: Imperative version of the map transformation rule. `barrier()` indicates a thread synchronization step. `f` is the mapping function. `q` is the sequentialization factor.

5.2.2 Reduce

```
Input: array xss[n][m]
Output: array of size [n]
---
map (λxs → reduce ⊕ 0⊕ xs) xss
≡
map (λxs →
   let ys = map (λx →
      reduce ⊕ 0⊕ x
   ) (split_{m/q} xs)
   in reduce ⊕ 0⊕ ys
) xss
```

Listing 11: High-level reduce transformation rule.

```
Input: array xss[n][m]
Output: array of size [n]
---
segmap(group, n, m):
   segred(thread) ⊕ 0⊕:
      let x = xss[gid, tid]
      in f x
≡
let m' = ⌈m/q⌉
segmap(group, n, m'):
   let tile = xss[gid, 0:m:1]
   let ys =
      segmap(thread):
         let idx = tid·q
         let chunk = tile[idx:q:1]
         in redomap ⊕ f 0⊕ chunk
   in segred(thread) ⊕ 0⊕:
      let x = ys[tid]
      in x
```

Listing 12: Intermediate reduce transformation rule. `f` is the mapping function of the **`segred`** in the original program.

The transformation rule for **`reduce`** in both formats can be seen in listings 11 and 12. It is assumed the array `xss` has shape `[n][m]`. As there is a dependency between the computation of earlier elements and later, the rule needs to produce code to handle this. As the inner **`reduce`** computes the reduction of `x` of size `q` we will end up having a pr. chunk reduction, which is saved in `ys`. Once this is done, the reduction across all chunks can occur using the same operator and neutral element. A rule showing the imperative semantics is shown in listing 10.

Note that in the intermediate rule, we also have a function `f` applied to the element `x` before it is reduced. This is the case since **`segred`** is just a perfect map nest on top of some bottommost **`redomap`** that applies the function `f` before reducing. For the same reason a **`redomap`** is used in the transformed program, so that `f` can applied. If `f` is simply the identity function, this would just be a normal reduction. Finally, a **`segred`** can be used to compute the reduction result across threads, but note here that `f` is not applied, since it already was on the intermediate results.

Work

The work complexity of the program is $W(\mathtt{nm})$ as we have `nm` elements to process. In the transformed program the first reduction happens on `x` of size `q`, but happens `m'` times giving $W(\mathtt{m})$ which happens `n` times, giving a final work complexity of $W(\mathtt{nm})$ as in the original program. As explained for the **`map`** rule, the reading of tiles and the sequential SOACs do not affect the work complexity.

Depth

For the high-level rule, the depth complexity is $D(\lg \mathtt{m})$ as each inner **`reduce`** performs a reduction on an array of size `m`. In the transformed, this is the case as well since the inner **`reduce`** works on a chunk of size `q` giving $D(\lg \mathtt{q})$. These, pr. chunk reduction results are then reduced, and as there are $\mathtt{m}/\mathtt{q}$ of these, this gives $D(\lg \mathtt{m}/\mathtt{q})$. Adding this up we finally have $D(\lg \mathtt{q} + \lg \mathtt{m}/\mathtt{q})$, but since `q` is considered constant this is $D(\lg \mathtt{m})$.

In the intermediate rule, the depth is also $D(\lg \mathtt{m})$, as this is the size of the segmented reduction. It is however now as clear for this rule that the depth is not altered by the transformation. In the first **`segmap`** each inner **`redomap`** works on a chunk of size `q`, but now sequentially giving $D(\mathtt{q})$. This gives the $\mathtt{m}/\mathtt{q}$ intermediate results that are reduced in parallel, giving $D(\lg \mathtt{m}/\mathtt{q})$. Adding this up we have $D(\mathtt{q} + \lg \mathtt{m}/\mathtt{q})$, but since `q` is constant, this is still $D(\lg \mathtt{m})$ meaning the depth is unaltered by the transformation.

Imperative `reduce` rule

```
Input: array xss[n][m],
       reduce operator ⊕,
       neutral element 0⊕
Output: array out[n]
---------------------------------------
forall gid = 0 to n do
   ac = 0⊕
   forall tid = 0 to m do
      let x = xss[gid, tid]
      ac = ac ⊕ (f x)
   out[gid] = ac
≡
let m' = ⌈m/q⌉
forall gid = 0 to n do
   let tile = xss[gid, 0:m:1]
   barrier()
   let tmp = scratch m'
   forall tid = 0 to m' do
      let idx = tid·q
      let chunk = tile[idx:q:1]
      tmp[tid] = redomap ⊕ f 0⊕ chunk
   barrier()
   ac = 0⊕
   forall tid = 0 to m' do
      let x = tmp[tid]
      ac = ac ⊕ x
   out[gid] = ac
```

Listing 13: Imperative version of the reduce transformation rule. `f` is the mapping function. `q` is the sequentialization factor.

5.2.3 Scan

```
Input: array xss[n][m]
Output: array of size [n][m]
---
map (λxs→ scan ⊕ 0⊕ xs) xss
≡
map (λxs →
  let ys = map (λx →
    reduce ⊕ 0⊕ x
  ) (split_{m/q} xs)
  let zs = scan_excl ⊕ 0⊕ ys
  in map2 (λx z →
    scan ⊕ z x
  ) (split_{m/q} xs) zs
) xss
```

Listing 14: High-level scan transformation rule. **map2** maps over two arrays of equal size.

```
Input: array xss[n][m]
Output: array of size [n][m]
---
segmap(group, n, m):
   segscan(thread) ⊕ 0⊕:
      let x = xss[gid, tid]
      in f x
≡
let m' = ⌈m/q⌉
segmap(group, n, m'):
   let tile = xss[gid, 0:m:1]
   let ys =
      segmap(thread):
         let idx = tid·q
         let chunk = tile[idx:q:1]
         in redomap ⊕ f 0⊕ chunk
   let zs =
      segscan_excl(thread) ⊕ 0⊕:
         let x = ys[tid]
         in x
   in segmap(thread):
      let d = zs[tid]
      let idx = tid·q
      let chunk = tile[idx:q:1]
      in scanomap ⊕ f d chunk
```

Listing 15: Intermediate scan transformation rule. `f` is the mapping function of the **segscan** in the original program.

The transformation rule for **scan** in both formats can be seen in listings 14 and 15. It is assumed `xss` has shape `[n][m]`. To successfully split the scan up between multiple chunks, each chunk needs to know the final result of the previous chunk to use as the accumulator. This can be achieved by first computing a reduction of each chunk which is stored in `ys`. This is, however, on a pr. chunk basis. To add the results of the previous chunks so that it becomes a running accumulator across chunks a **scan** (exclusive) is performed, which then computes the starting accumulator of each chunk. Finally, the chunks can be scanned with this accumulator. A rule showing the imperative semantics is shown in listing 10.

It should be noted how the **scan** transformation rule presented here differs slightly from the sequentialized intra-block version shown in figure 4. Rather than doing an initial reduction using **redomap** as in the first **segmap** at the thread level, a scan could also be performed using a **scanomap** instead. This way no scan would have to be done in the final **segmap** which could save some computations of applying `f`. If

f is some expensive computation, this would be the preferred implementation. However, when f is small, i.e. a simple arithmetic operator, this version did not show any performance improvement and compiled to CUDA code that used more registers. For this reason, and since the version shown in listing 15 allowed for code reuse in the eventual implementation, see section 7.2.3, this was chosen.

Work

The work complexity of the original programs is $W(\mathtt{nm})$ as we have nm elements to process. In the transformed program, the first **map**-**reduce** nest has $W(\mathtt{m})$, the intermediate scan $W(\mathtt{m'})$ and the final **map**-**scan** nest has $W(m)$, as m' depends on m, adding all these up we have $W(m)$, but it happens n times giving a final work complexity of $W(nm)$.

Depth

For the high-level transformation rule, the depth complexity of the original program is $D(\lg \mathtt{m})$ as each inner **scan** process m elements. For the transformed program we have the **map**-**reduce** nest has a depth of $D(\lg \mathtt{q})$ as each reduction reduces q elements. The intermediate **scan** has a depth of $D(\lg \mathtt{m}/\mathtt{q})$, and finally the **map**-**scan** nest has a depth of $D(\lg \mathtt{q})$. Adding ll these up gives $D(\lg \mathtt{q} + \lg \mathtt{m}/\mathtt{q} + \lg \mathtt{q})$, but since q is constant this is $D(\lg \mathtt{m})$.

For the intermediate transformation rule, the depth complexity of the original program is $D(\lg \mathtt{m})$ as each inner **scan** process m elements. We know from the analysis of **reduce** that the depth of the sequential **redomap** is $D(\mathtt{q})$. The depth of the intermediate scan is $D(\lg \mathtt{m}/\mathtt{q})$ and the depth of the sequential **scanomap** depth is also $D(\mathtt{q})$. Adding these up we have a depth of $D(\lg \mathtt{m} + \mathtt{q} + \mathtt{q})$, but as q is constant, we still have a depth of $D(\lg \mathtt{m})$.

Imperative `scan` rule

```
Input: array xss[n][m],
     scan operator ⊕,
     neutral element 0⊕
Output: out[n][m]
```

```
forall gid = 0 to n do
   let ac = 0⊕
   forall tid = 0 to m do
      let x = xss[gid, tid]
      ac = ac ⊕ (f x)
      out[gid, tid] = ac
≡
let m' = ⌈m/q⌉
forall gid = 0 to n do
   let tile = xss[gid, 0:m:1]
   barrier()
   let ys = scratch m'
   forall tid = 0 to m' do
      let idx = tid·q
      let chunk = tile[idx:q:1]
      ys[tid] = redomap ⊕ f 0⊕ chunk
   barrier()
   let zs = scratch m'
   let ac = 0⊕
   forall tid = 0 to m' do
      zs[tid] = ac
      let x = ys[tid]
      ac = ac + x
   barrier()
   forall tid = 0 to m' do
      let d = zs[tid]
      let idx = tid·q
      let chunk = ys[idx:q:1]
      tile[tid] = scanomap ⊕ f d chunk
   barrier()
   out[gid, 0:m:1] = tile
```

Listing 16: Imperative version of the scan transformation rule. `Barrier()` indicates a thread synchronization step. `f` is the mapping function. `q` is the sequentialization factor.

5.2.4 Scatter

```
Input: array xss[n][m]
Output: array of size [n][m]
---------------------------------------
map (λ ds is vs →
   scatter ds is vs
) dss iss vss
≡
map (λ ds is vs →
   map (λ i v →
      scatter ds i v
   ) (split_{m/q} is vs)
) dss iss vss
```

Listing 17: High-level scatter transformation rule.

```
Input: array xss[n][m]
Output: array of size [n][m]
---------------------------------------
segmap(group, n, m):
   let ds = dss[gid, 0:m:1]
   segmap(thread):
      let i = iss[gid, tid]
      let v = vss[gid, tid]
      in ds with [i] = v
≡
let m' = ⌈m/q⌉
segmap(group, n, m'):
   let ds = dss[gid, 0:m:1]
   let is = iss[gid, 0:m:1]
   let vs = vss[gid, 0:m:1]
   loop ds' = ds
   for j = 0 < q do:
      segmap(thread):
         let idx = tid·q+j
         let i = is[idx]
         let v = vs[idx]
         in ds' with [i] = v
```

Listing 18: Intermediate scatter transformation rule. `q` is the sequentialization factor.

The transformation rule for **`scatter`** can be seen in listings 17 and 18. Here we have three arrays. First is `dss` of shape `[n][k]` which is the destination array of the **`scatter`**. Second is both `iss` and `vss` of shape `[n][m]`, where `k` might be different from `m`. Note that in the transformed program, the inner **`map`** is only over the chunks of `is` and `vs`, as we still want to be able to scatter to all locations within the original `ds` and not just a chunk of it. A rule showing the imperative semantics is shown in listing 19.

There is no actual segmented scatter operator present in the intermediate representation. Instead a **`scatter`** is represented with a **`segmap`** that does an in-place update instead of returning a value. Also note that in the intermediate rule in the original program, `ds` is already implicitly a tile of `dss`, so in the transformed program we only need to tile the remaining two. Now, as we do not know to which indices the scatter is going to write, we have to update the array using a sequential loop with `q` iterations, where each iteration of a thread reads a new element of `iss` and `vss` to scatter to the array.

Work

The work complexity of the original program is $W(\mathtt{nm})$. The inner size `k` of `dss` does not affect the complexity. For the transformed program, the analysis is much the same as for **`map`**. Each inner **`scatter`** now processes `q` elements, this is done `m'` times which is done a total of `n` times across all groups. As such

the work complexity of the transformed program is also $W(\texttt{nm})$.

Depth

The depth complexity of the original program is $D(1)$. Each thread reads a single index and value and writes it to the output array. The same can be said for the transformed program in the high-level rule. However, in the intermediate rule we now have a loop representing the sequential part, rather than a SOAC as in previous rules, which wraps the inner **`segmap`**. But even here the same logic applies in that `q` is considered constant, meaning the loop executes a constant amount of times. It is not dependent on the input size.

Imperative `scatter` rule

```
Input: array dss[n][k], iss[n][m], vss[n][m]
Output: dss[n][k]
---------------------------------------------

forall gid = 0 to n do
   forall tid = 0 to m do
      let i = iss[gid, tid]
      let v = vss[gid, tid]
      dss[gid,i] = v

≡
let m' = ⌈m/q⌉
forall gid = 0 to n do
   let ds = dss[gid, 0:m:1]
   let is = iss[gid, 0:m:1]
   let vs = vss[gid, 0:m:1]
   forall tid = 0 to m' do
      forall j = 0 to q do
         let idx = tid·q+j
         let i = is[idx]
         let v = vs[idx]
         ds[i] = v
```

Listing 19: Imperative version of the scatter transformation rule. `Barrier()` indicates a thread synchronization step. `q` is the sequentialization factor.

6 Compiler Internals

While the rules of the previous sections serve their purpose well for explaining how the transformations work both at a high abstraction and also at a lower level, the internals of the compiler are a bit more abstract and have slightly different semantics. This section will go over how the different concepts discussed so far map to internal structures in the compiler and where their semantics differ.

6.1 SOACs

The semantics used for the SOACs, both normal and segmented, so far do not match exactly the semantics of the ones used in the compiler. The compiler uses a *tuple-of-arrays* representation for arrays of tuples, meaning all SOACs accept multiple parameters and can produce multiple values. Further, **redomap** and **scanomap** both accepts multiple binary operators and can produce both *reduced/scanned results* and *mapped results*. For some binary operators $\overline{\oplus} = \oplus_1 \cdots \oplus_k$ with corresponding neutral elements $\overline{0_\oplus} = 0_{\oplus_1} \cdots 0_{\oplus_k}$ and some arrays $\overline{\texttt{xs}} = \texttt{xs}_1 \cdots \texttt{xs}_j$ for some $j \geq k$, the semantics of **redomap** can be understood as;

$$
\begin{array}{l}
\textbf{redomap}\ \overline{\oplus}\ \texttt{f}\ \overline{0_\oplus}\ \texttt{xs} \\
\qquad\qquad \Downarrow \\
\textbf{let}\ \texttt{x}_1\ \ldots\ \texttt{x}_k\ \ldots\ \texttt{x}_j\ =\ \textbf{map}\ \texttt{f}\ \overline{\texttt{xs}} \\
\textbf{let}\ \texttt{y}_1\ =\ \textbf{reduce}\ \oplus_1\ 0_{\oplus_1}\ \texttt{x}_1 \\
\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \\
\textbf{let}\ \texttt{y}_k\ =\ \textbf{reduce}\ \oplus_k\ 0_{\oplus_k}\ \texttt{x}_k \\
\textbf{in}\ \texttt{y}_1\ \ldots\ \texttt{y}_k\ \texttt{x}_{k+1}\ \ldots\ \texttt{x}_j
\end{array}
$$

That is, the **map** produces j different elements, of which the first k are passed to the binary operators in order. The remaining $j - k$ results are simply returned as one of the mapped results. The same applies to **scanomap**. The semantics of **segred** and **segscan** are altered accordingly as they are still a perfect map nest on top of either a **redomap** or **scanomap**. Note that when there is only a single binary operator and neutral element, it has the same semantics as when used in the rules.

6.2 Intermediate Representation

At the most basic level, the *intermediate representation* (IR) is a textual representation of the abstract syntax tree of the program. This representation is morally identical to the one introduced in section 5.1.2, but with the new semantics, meaning that segmented operators are used to express parallel operations and it is these that represent the kernels that will ultimately be generated by the compiler. We will spend some time going over the most essential data structures inside the compiler that are needed to apply the optimization.

The `SegOp` type, representing a segmented operator and seen in listing 20, is generic over some types `lvl` and `rep`. The `rep` type is not that important for the transformations as they do not change the internal representation of the program. Just know that it is set to represent that we are working at GPU representation. The `lvl` type is to represent at which hardware level the segmented operator is to be mapped. Since the GPU has two levels, this can either be the block level (named Group inside the compiler)

or the thread level. For this project, the `lvl` type is always set to that of `SegLevel` which has two constructors, mapping the `SegOp` to either hardware level. With all this, the concrete type of the segmented operators is `SegOp SegLevel GPU`.

```
data SegOp lvl rep
  = SegMap lvl SegSpace [Type] (KernelBody rep)
  | SegRed lvl SegSpace [SegBinOp rep] [Type] (KernelBody rep)
  | SegScan lvl SegSpace [SegBinOp rep] [Type] (KernelBody rep)
  | SegHist lvl SegSpace [HistOp rep] [Type] (KernelBody rep)
  deriving (Eq, Ord, Show)
```

Listing 20: Type definition of a `SegOp`

The `SegSpace` type encodes the shape of the segmented operators. Say if we had a kernel operating on two dimensional array, then the outer `SegOp` at group level would encode the first dimension, and inner `SegOps` the inner dimensions. The `SegBinOp` type represents the binary operator of the segmented operator, which is why it is only present for `SegRed` and `SegScan`. Note that it is an array of `SegBinOps` as there can be multiple operators applied. Also, note that despite the name, a single `SegBinOp` might take more than two arguments due to being fused. The `Type` type represents the types of the results produced by the `SegOp`, and as it can produce multiple results this is an array. Finally, the `KernelBody` represents the body of the kernel produced. It is here it will read the values from different arrays, maybe do some prepossessing on them, before handing them off to the `SegBinOps`.

Another internal compiler construct of significance to this project is the `SOAC` type and specifically its value constructor `Screma`. A single `Screma` represents a combination of scans, reduce, and maps. A `Screma` contains a `ScremaForm` that describes its functionality. It is illustrated in listing 21. This construct allows the new semantics described earlier. The `Lambda` is the map part, and then it can contain any number of scans and reductions. That is, if it contained some map function and some reductions then it would represent a **`redomap`** , while if the `Lambda` was the identity function and it contained only one `Scan` then it would have the semantics of a scan as described at the beginning of the report.

```
data ScremaForm rep
  = ScremaForm
      [Scan rep]
      [Reduce rep]
      (Lambda rep)
  deriving (Eq, Ord, Show)
```

Listing 21: Type definition of a `ScremaForm`

The semantics of a `SegOp`, or more precisely the two `SegOps` to contain `SegBinOps` have very similar semantics to that of a `Screma` except they are exclusive to only do scans or reductions. Each `SegBinOp` takes two or more arguments, some of which are the neutral element/accumulator and they are fed the remaining arguments from the `KernelBody` which might produce multiple values.

With these similar semantics between `SegOps` and `Scremas`, and the fact that they both mostly consist of inner `Lambdas` makes it fairly trivial to convert from `SegOp` to `Screma` which is what we want for the sequential parts illustrated in the transformation rules.

We have the structure for an actual statement within the IR reproduced in listing 22. The `stmPat` contains the left-hand side of the binding along with information on which type each bound variable in the pattern should have. `stmExp` represents the expression that produces the values to be bound to the pattern. All the segmented operators can be used as an expression, but we also have basic expressions for arithmetic, loops, branches and so on.

```
1 data Stm rep = Let
2   { -- | Pat.
3     stmPat :: Pat (LetDec rep),
4     -- | Auxiliary information statement.
5     stmAux :: StmAux (ExpDec rep),
6     -- | Expression.
7     stmExp :: Exp rep
8   }
```

Listing 22: The structure of a statement in the IR is a let-binding with type information and an expression.

The Futhark compiler has a set of core representations of which we are interested in the ones related to GPU code generation, namely the GPU representation and the GPUMem representation. In the GPU representation flat segmented operations, as those from section 5.1.2, are used to express parallelism [19] and can be considered a pre-stage to the GPUMem representation as this is where memory information is added. For this project, it was decided to work on the GPU representation, as this is considerably more straightforward than the GPUMem representation and it should be sufficient for demonstrating the effects of the optimization. Furthermore, while the optimization is concerned with how data is stored in memory, i.e. operating on shared memory as opposed to global memory and keeping intermediate thread local results in registers, this can still be somewhat controlled at the GPU representation. Results of segmented operators can be instructed to be kept in registers, and since we are in the intra-group version intermediate arrays can be allocated in shared memory [11].

7 Implementation

This section will cover the most interesting implementation-specific details of our transformations. Some of these details are not specific to the transformation of any specific operation but are something the implementation needs to be aware of. Next, the specific transformations for each operator will be explained. Finally, we will describe how our optimization decides whether it should apply itself to the program, both at compile time and at run time.

7.1 Optimization Passes

The Futhark compiler can roughly be broken down into three parts: frontend, middle-end, and backend. The frontend is responsible for parsing the source language, type-checking, and generating the core IR which is then passed to the middle-end. The middle-end transforms the IR through a set of *optimization passes*, based on the backend chosen. This middle-end is the part of the compiler we concern ourselves with, as this is where the optimization will be added. Finally, the backend is responsible for the final code generation based on the IR passed from the middle-end. [19]

In short, a compiler pass is a pure function that when given a program as input produces another program as output. Since the Futhark compiler is heavily optimizing and can produce code for multiple backends with different hardware specifics, this is a convenient way of structuring the architecture as it enables constructing *pipelines*. A pipeline is a composition of passes that dictates how an IR program should be transformed into another IR program. This makes it easy to extend the compiler with new transformation techniques, as one can extend a pipeline with the given transformation.

Our optimization has been implemented as a single module in the file `IntraSeq.hs`, which exposes a function that is added as a pass to the GPU pipeline. The implementation relies on two other passes, namely `simplifyGPU` and `unstreamGPU`. The `simplifyGPU` pass does what the name implies, including removing dead bindings. This simplifies the implementation of our pass, as then we can create statements that might or might not be used later on, letting the `simplifyGPU` pass remove them if they are not. For the code that is to be executed sequentially, we use the built in SOACs as these are simple to work with but they cannot be translated to actual GPU code. As such the `unstreamGPU` pass is used to convert these SOACs into loops which can be translated. Ultimately, the required pipeline for running the `IntraSeq` optimization is seen below.

simplifyGPU: It is assumed a `simplifyGPU` runs before our optimization pass. This is mostly for precaution to make sure that there a no leftover statements from an earlier pass that the code might not know how to handle and fail when it could have produced a working program.

intraSeq: This is our optimization pass. As we shall see shortly this will potentially generate redundant code which makes the implementation more straight forward.

simplifyGPU: Another simplify pass is added after our to ensure that the program is left in a clean state, with no dead bindings and redundant computation before the next pass takes over.

unstreamGPU: This pass is needed to convert the generated `Screma` to sequential loops. Later on in the compilation process, Futhark will throw an error if any SOACs remain.

7.2 Code generation

This section will outline the main implementation details of the transformations. There are multiple common transformations regardless of the operator which will be introduced first, after which details of each operator will be looked at.

During the transformation, some information about the context in which the transformation is happening is needed. This is encoded in the `Env` data type shown in listing 23. The two variables, `grpId` and `threadId`, are equivalent with CUDA `blockIdx.x` and `threadIdx.x`, and are needed to know from where data should be read. The group size of the original `SegOp` and the updated one after sequentialization are stored in `grpSize` and `grpsizeOld` respectively. `nameMap` is a mapping from arrays used in the original program to the tiles produced for them. Finally, the sequentialization factor decided by the programmer is stored in `seqFactor`. `seqFactor` is the same value that is denoted as `q` in the transformation rules and pseudocode presented in this thesis.

```
1 data Env = Env {
2   grpId      :: SubExp, -- The group id
3   grpSize    :: SubExp, -- The group size after sequentialization
4   grpsizeOld :: SubExp, -- The group size before sequentialization
5   threadId   :: Maybe VName, -- The thread id if available
6   nameMap    :: M.Map VName VName, -- Mapping from arrays to tiles
7   seqFactor  :: SubExp -- The sequentialization factor
8 }
```

Listing 23: Environment data type

7.2.1 Tiles and Chunks

Tiling is the process of copying data from global memory to shared memory, such that a kernel has access to all the data it needs in shared memory. This is an important step in the transformations to ensure that the number of accesses to the slow global memory is minimized. When introducing sequentialization, care must be taken with regard to the access pattern. Before the sequentialization transformation has been applied, the kernel has implicit coalesced access to global memory as each thread simply accesses a single element, and all adjacent threads access adjacent elements. This is however not necessarily the case after the transformation has been applied. If a thread is simply instructed to read consecutive elements into shared memory then the access will not be coalesced as the threads in the same warp will then access elements with a stride. Instead, all threads will first cooperate to move the data from global to shared memory, as illustrated in figure 5. Once in shared memory, the threads are free to read a chunk of consecutive elements to process.

Another important job of the tiling code is to ensure that once an array is loaded into shared memory, it is padded to ensure its size is a multiple of the sequentialization factor, this is also illustrated in figure 5. This is to avoid any potential out-of-bounds access and will be discussed in more detail.

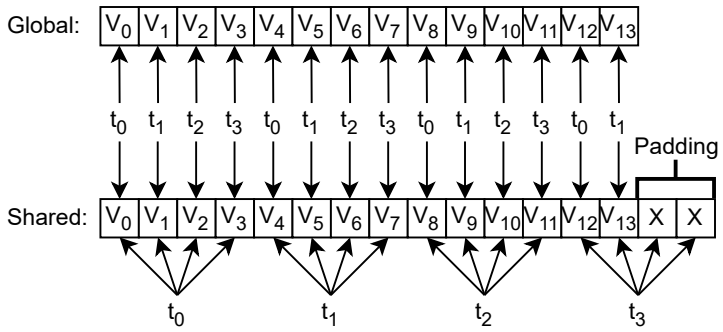


Figure 5: Coalesced access to global memory for 4 threads with a sequentialization factor of 4. In a single access, the warp with threads $t_1, \ldots t_4$ accesses consecutive elements in global memory. From shared memory, each thread can then read its chunk of consecutive elements.

The actual IR code generated to allow for this access pattern is fairly simple as by using the internal compiler construct for slices, Futhark will already compile this to a coalesced access pattern. The pseudo-IR of listing 24 illustrates the IR generated by the optimization pass. An intermediate scratch array of size `tile_size`, which is the size of the data rounded up to the closest multiple of the sequentialization factor, `q`, is allocated. Then the data is read from the global array `a` and is put in the `scratch` array. As these statements are at the group level Futhark knows how to use the threads available to read the data coalesced based on the slice. Finally, the **`segmap`** at the thread level is used to get each thread to read their respective chunks of the tile and save them into registers as indicated by the `private` annotation.

```
let m' = ⌈m/q⌉
let tile_size = m'·q

segmap(group, n, m'):
  ...
  let tile_scratch = scratch(tile_size)
  let slice = a[gid, 0:m:1]
  let staging = (tile_scratch with [0:m:1] = slice)
  let tile =
    segmap(thread):
      let start = q·tid
      in private chunk = stagning[start:q:1]
  ...
```

Listing 24: Pseudo tiling code generated for an array `a:[m][m]`. `q` is the sequentialization factor.

The optimization also has to determine which arrays to create tiles for by identifying the arrays read that are currently not stored in shared memory. To determine this, we take advantage of the invariant that once we encounter `SegMap` at group level, i.e. signaling the start of a new kernel, then all arrays in scope must necessarily be located in global memory and should be tiled. If tiling code is generated for an array that is unused in the kernel, then the `simplifyGPU` pass can be relied on to remove it.

Transposed Tiles

Initially, the tiling code did not work for transposed arrays. Once a source Futhark program that uses transpose has been compiled to IR code, the transposition is implicit, as all future accesses to the transposed arrays just have their indices flipped. This means that during the generation of the tiling code, it has to be detected if the array being tiled has in fact been transposed. For two-dimensional arrays, this can be done by looking at the shape of the `SegMap` at the group level and comparing it to the type of the array. If we have some array `a` of shape `[n][m]`, but the `SegMap` is mapped across the `m` dimension, then it means the array has been transposed, and we need to fix the indices accordingly.

7.2.2 Reshaping results

When applying the optimization on a kernel it will inevitably change the return type of that kernel. Assume we had an array `a` of size `[n][m]`. This will result in a kernel with n blocks with m threads each before sequentialization and $m' = \lceil m/q \rceil$ threads after. However, after sequentialization, each thread now potentially produces arrays rather than single values which causes type errors. The problem is illustrated below in listing 25 where the pseudocode has been extended with type annotations. In the original, a **`segmap`** at thread level produced an intermediate array of type `[m]`α, but after sequentialization, it now produces an intermediate array of type `[m']`[q]α. This gives a type error, but it is not possible to simply change the type of `xs` since it might be used by statements later in the program, so a pass over all statements would be required to adjust this. Instead, it is ensured that the type of the original bound variable remains unchanged by first saving the result of the inner **`segmap`** in `zs` which can then be flattened to the original and correct size before returning.

```
let xs : [n][m]α = segmap(group, n, m):
   let ys : [m]α = segmap(thread):
                           <body>
   in ys

      ⇓
let xs : [n][m]α = segmap(group, n, m'):
   let zs : [m'][q]α = segmap(thread):
                           <optimized body>
   let ys : [m]α = flatten zs
   in ys
```

Listing 25: IR showing how the flattening of results is used to keep the program type safe.

7.2.3 Transformations of SOACs

This section will present an outline of the implementation-specific details for each of the operators. The implementation is largely a direct mapping of the intermediate-level rules presented in section 5.2 and thus focus will be on the specific parts that deviate from these.

Common Transformations

There are a couple of common transformations that need to be applied to all the operators. First, the `SegSpace` has to be modified to reflect that we now have $m' = \lceil m/q \rceil$ threads, where m was the original amount of threads.

Secondly, if any statement inside of a sequentialized `SegOp` uses the original `tid` in some computation, then this must be replaced by the new thread ID multiplied by the index of the element in its chunk the thread is currently processing. To see this, look at the following example using **`iota`**[1]operator. Much like with transpose, the usage of **`iota`** can be made implicit in the generated IR without any explicit invocation of **`iota`**. An example is illustrated in listing 26 which maps over an array `xss` of shape `[n][m]` and uses **`iota`** inside of the outer map to add the values $0, \ldots, m-1$ to each respective value of `xs`. This program, when translated to an intra-group kernel in the IR is equivalent to the one in listing 27. As can be seen, the threadID `tid` of the inner **`segmap`** is used in place of the reading an actual value from some **`iota`** array. This optimization saves on the memory used by the program but causes the program to produce incorrect results after sequentialization since each thread now processes consecutive elements. This means that thread with ID 0 should add the values, 0, 1, 2, and 3 to its elements, the thread with ID 1 should add 4, 5, 6, and 7 to its elements, and so on.

```
1 let xss' = map (\ xs ->
2     let xs' = map2 (\x i -> x + i) xs (iota m)
3     in xs'
4 ) xss
```

Listing 26: Simple Futhark program utilizing **`iota`**

```
segmap(group, n, m):
   segmap(thread):
      let x = xss[gtid, tid]
      let res = x + tid
      in res
```

Listing 27: IR code generated for the program from listing 26

To solve this issue, an **`iota`** over the sequentialization factor, q, is created producing an array of the values $0, \ldots, q-1$ which each thread can then use to get the correct offset. If unused, the `SimplifyGPU` pass will remove it.

Map

The implementation of the **`map`** rule essentially follows the transformation rule. The body of the **`segmap`**, namely the statements that read elements and the application of the map function `f`, is split up. The latter is converted to a sequential map, and the former is now replaced with statements that read from the respective tile instead of directly from the input array.

[1] **`iota`** is a function that takes an integer n and returns an array of consecutive integers from $0, \ldots, n-1$

Reduce

The implementation of the **reduce** rule does not follow the transformation rule exactly due to technical limitations within the compiler. As mentioned, if the size of the array is not a multiple of the sequentialization factor $\mathtt{q}$ then the last thread might have fewer elements to process. At a glance, it would seem simple to just have each thread compute the size of its chunk to be reduced and simply perform a reduction of that size. However, Futhark does not currently support this kind of irregularity of the threads, resulting in a compiler error. This implies that a thread that does not have $\mathtt{q}$ elements to process sequentially will still have to loop $\mathtt{q}$ times. This is no issue for the other operators since padding is added to the tiles, so the threads that do not have $\mathtt{q}$ elements to process will simply produce some unused results. However, since a reduction produces a single value as its result, these excess computations will corrupt the reduction result of the thread that does not have $\mathtt{q}$ elements to process. A straightforward way to handle this is to simply generate a **scanomap** instead of the **redomap**, utilizing that a **scan** is essentially the same as a **reduce** that just produces all the intermediate results. Each thread then simply has to return the result of the **scanomap** that corresponds to its final reduction result. This index can be calculated with some simple arithmetic instructions to avoid the use of a branching instruction.

For example, since a thread now uses a **scanomap** instead of a **redomap**, it produces an array of size q instead of a single result. To find the element in the array of scan results that corresponds to the last valid reduction result for a thread that has processed q' elements sequentially, its reduction result would be that at index $[q' - 1]$ of the **scanomap** result.

Pseudo IR code representing the actual implementation of the **reduce** transformation rule is given in listing 28.

```
segmap(group, n, m):
   segred(thread) ⊕ 0⊕:
      let x = xss[gid, tid]
      in f x
≡
let m' = ⌈m/q⌉
segmap(group, n, m'):
   let tile = xss[gid, 0:m:1]
   let ys =
      segmap(thread):
         let idx = tid·q
         let chunk = tile[idx:q:1]
         let tmp = m - idx
         let size = min(q, tmp)
         let redi = size - 1
         let res = scanomap ⊕ f 0⊕ chunk
         in res[redi]
   in segred(thread) ⊕ 0⊕:
      let x = ys[tid]
      in x
```

Listing 28: Reduce transformation implementation.

Since the **scanomap** is produced at the level where each thread is operating sequentially inside of registers, there is not much downside to this in terms of execution time as opposed to producing a **redomap** besides having to do the additional arithmetic instructions. However, the scan does require additional registers which could potentially cause some slow-down on large workloads.

Scan

Similar to **map**, the implementation for **scan** closely follows the intermediate-level transformation rule. The **segscan** is essentially split up into three steps. First, a reduction from which code can be reused from the **reduce** implementation. This also means that similar to **reduce**, the **scan** implementation generates a **scanomap** instead of a **redomap** in this step. A **segscan** is then generated to perform a blockwide scan over the reduction results which can then finally be used in a sequential scan through a **scanomap** to generate the final scan results.

Scatter

The implementation of **scatter** follows the transformation rule closely. However, extra care has to be taken when the size `m` is not a multiple of sequentialization factor `q`. If this is the case, the chunks of the last thread, i.e. the indices and values it has to write, will be padded with undefined values. This means that if a thread detects that the current loop counter is larger than the number of actual elements in its chunk, it should not perform the write. As such if this is detected it will simply write the value at the index it did the

previous iteration. This is not a problem, since if multiple threads were to write to the same position in the output array, it is undefined which one will end up there.

Another thing worth mentioning is how a **`scatter`** operation is detected. There is no segmented **`scatter`** operation, but rather just a map not directly returning any values but performing an update. This means that each time a **`segmap`** is encountered, we first have to inspect the return statements of the operation to see if it is a **`segmap`** or if it is a **`scatter`** in disguise. From testing different combinations of maps and scatters, we were not able to get the compiler to fuse any such **`segmap`** with the different return functions. This means our implementation assumes the return statements will always be one kind or the other.

7.3 Conditional Optimization

The optimization is only going to target the intra-group kernels of the program. As such the first step of the implementation should be to identify which of the generated code versions is the intra-group one and apply the transformation there. However, to focus on the task at hand we have implemented a new attribute called `seq_factor(x)`. Attributes allow the programmer to annotate their programs in order to instruct the compiler to perform specific tasks. In the `seq_factor(x)` attribute, `x` can be any positive integer and determines the sequentialization factor to use during the code transformation. This attribute should be applied to a **`map`** in the source language along with the attribute `incremential_falttening(only_intra)` to enable only the generation of intra-group kernels. See listing 29 for an example. This attribute has to be present for our optimization to be performed. This means if applied to something that will not be an intra-group kernel it is considered a user error. This should of course be changed if the optimization is to be merged into the main Futhark compiler, but allowed for easy experimentation during development. Besides this attribute, a series of other properties has to be fulfilled which can be divided into compile-time properties and run-time properties.

```
1 #[incremental_flattening(only_intra)]
2 #[seq_factor(4)]
3 map (\ xs ->
4     let ys = scan (+) 0 xs
5     in reduce (+) 0 ys
6 ) xss
```

Listing 29: Program which will only generate an intra-group code version and to which our implementation will be applied with a sequentialization factor of 4.

7.3.1 Compile time

Besides requiring the `seq_factor` attribute to be set, there is also a restriction on the shape of the input.

The shape of the input has to be two-dimensional. Assume we had an array `a` with dimensions `[n][m][k]` and that the sequentialization factor is 4. The outer dimension of size `n` would be mapped to blocks in the hardware, and the remaining two to the threads. It is however not entirely clear what should happen with the remaining two dimensions when applying the transformation. Should it sequentialize over the dimension

of `m` meaning each thread would now process 4 arrays each of size `k` or should you maybe flatten out the inner dimension to be `[nk]`? For the former, if `k` is big, this is probably not what we want as then each thread would have to sequentially process $4 \cdot$ `k` elements each. For the latter, the issues are most clear with an example.

Assume the shape of `a` is `[n][5][3]`, a sequentialization factor of 4, and that we want to reduce across the third dimension. If `a` is flattened to have the shape `[n][5*3]`, then the program still spawns `n` blocks, each with $\lceil 15/4 \rceil = 4$ threads. Each of these threads will process 4 sequential elements, but the inner dimension was only 3. As such, all threads will not have access to the 3 elements they need to perform their reduction. This is illustrated in figure 6, where the brackets denote the elements distributed to each thread. Consecutive elements of the same color should be reduced, but as is clear from the figure, not all threads have access to the elements needed to perform the reduction.

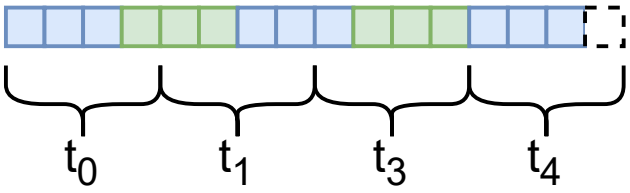


Figure 6: Figure illustrating the issues with flattening the inner two dimensions of a three dimensional array.

If the `seq_factor` attribute is set, but the input shape is not two-dimensional, the compiler will not give an error but resort to compiling without the efficient sequentialization optimization.

7.3.2 Run time

Even if the optimization is successfully applied to some program, it might not be more efficient than the original one. As such, in the spirit of the incremental flattening framework already present in the compiler, it will create two code versions and discriminate between them at run time. The transformation is illustrated in listing 30 below. That is, after the transformation has been applied a conditional statement is inserted to discriminate between the two versions at run time. In the condition `T` $\leq$ `m'`, `T` denotes some fixed threshold of threads that need to be used in the optimized program before it chooses this. Otherwise, it will just execute the original program. Were `T` to be set to e.g. 64, the optimized version would only run when using 64 or more threads.

```
segmap(group, n, m):
   <body>
≡
let m' = ⌈m/q⌉
if T ≤ m' then
   let x = segmap(group, n, m'):
             <optimized body>
   in x
else
   let x = segmap(group, n, m):
             <body>
   in x
```

Listing 30: Transformation showing how the two generated code versions are inserted.

8 Experimental Results

This section will examine the performance results of applying the efficient sequentialization optimization presented to a series of programs exhibiting intra-group parallelism.

8.1 Method

A comprehensive suite of small Futhark programs using the four parallel operators, **`map`**, **`reduce`**, **`scan`**, and **`scatter`**, has been developed with a twofold purpose, both serving as correctness tests and for benchmarking the performance of the transformations. A small subset of these were then selected to demonstrate the effects of the transformation for each operator. The initial benchmark results for a larger portion of the benchmark suite, once the transformation for each operator was implemented, can be seen in appendix F. Finally, to show the effects of the transformation on 'real' programs, benchmarks are presented of big-integer addition and intra-block radix sort.

The tests can be reproduced by using Futhark's built-in testing and benchmarking functionality [5]. First, clone and build the repository containing the modified compiler, then clone the repository containing the test suite, see section 1.2. Using `cabal` [8] a given benchmark can then be run from within the compiler project folder with the command:

```
cabal run futhark -- bench --backend=cuda ../path/to/benchmark.fut
```

As the optimization only targets intra-group kernels, the results will be compared to the run time of the equivalent intra-group kernel without efficient sequentialization applied. To ensure that only the intra-group versions are run, all tests are annotated appropriately with the `incremental_flattening` and `seq_factor` attributes. Finally, the goal of this section is to present the overall results in terms of applying efficient sequentialization, but not fine-tuning the performance of each specific program. A sequentialization factor of 4 has been chosen for all benchmarks. This value was chosen empirically, as it generally

showed consistent improvements, balancing the trade-offs of increasing the amount of sequential work. Finally, all values used are 32-bit.

For the experiments, given the execution time of a program without the optimization applied, et, and the execution time of the same program with the optimization applied, et', the speedup was calculated as:

$$speedup = \frac{et}{et'}$$

and the bandwidth was calculated as:

$$bandwidth(GB/s) = \frac{(b_r + b_w)/et_\mu}{1000}$$

where b_r is the number of bytes read from global memory by the program, b_w is the number of bytes written to global memory, and et_μ is the execution time in microseconds.

All tests are run on an NVIDIA A100 [15] with 80GB VRAM offering up to 2 TB/s of bandwidth.

8.2 SOACs

This section will examine the performance gains of applying the transformations to each of the four SOACs with minimal other factors influencing the results. Like the small programs presented in section 3.1, each test consists of a single outer **`map`** with the specific SOAC as the inner function, meaning the only code that is executed is the tiling code and the actual SOAC computation. As the transformation is applied on a per-SOAC basis, this will give the most clear indication of the performance gains of each of them.

Two different test setups have been created. For the first one, we keep a constant inner dimension of 1024, meaning the intra-group code version will run with 1024 threads in the original program and 256 in the optimized one. The purpose of this test is to see the impact of having more blocks doing the same amount of work. The second test keeps the workload constant but distributes it differently across blocks and block sizes. With this test, it should be possible to see if having more blocks with fewer threads or fewer blocks with more threads is most desirable. As an added benefit, for the version without sequentialization, the inner dimension is at most 1024 elements due to the CUDA cap of 1024 threads per block, but for the sequentialized version, larger sizes can be explored as each thread now processes multiple elements.

The results of both test setups can be seen below in table 4 and 5.

Number of blocks		100k	200k	300k	400k	500k
map	org (GB/s)	1260.53	1268.15	1268.78	1267.41	1267.68
	seq (GB/s)	1359.4	1368.37	1374.45	1377.69	1378.16
	speedup	1.0784	1.0790	1.0832	1.0870	1.0871
reduce	org (GB/s)	366.98	369.41	370.45	370.62	371.00
	seq (GB/s)	977.43	942.67	925.6	920.62	914.08
	speedup	2.6634	2.5518	2.4986	2.4839	2.4638
scan	org (GB/s)	421.67	420.16	420.08	420.35	420.33
	seq (GB/s)	825.99	808.12	802.79	798.91	799.40
	speedup	1.9588	1.9233	1.9110	1.9005	1.9018

Table 4: The effective bandwidth in GB/s and speedup for all the SOACs with 1024 threads pr. block. The *org* row is the bandwidth before optimization and the *seq* row after. All tests are with 32-bit values.

Sizes		[1048576] [128]u32	[524288] [256]u32	[262144] [512]u32	[131072] [1024]u32	[65536] [2046]u32	[32768] [4096]u32
map	org (GB/s)	1311.27	1327.95	1318.81	1268.12	-	-
	seq (GB/s)	1254.16	1358.17	1361.12	1357.26	1349.49	1320.84
	speedup	0.9564	1.0227	1.0320	1.0702	-	-
reduce	org (GB/s)	552.34	523.84	450.46	368.78	-	-
	seq (GB/s)	684.95	1026.37	1008.49	954.75	880.67	753.93
	speedup	1.2400	1.9593	2.2387	2.5889	-	-
scan	org (GB/s)	420.54	422.06	422.14	422.15	-	-
	seq (GB/s)	820.32	840.86	855.08	814.75	728.21	597.40
	speedup	1.9506	1.9922	2.0255	1.9300	-	-

Table 5: The effective bandwidth and speedup of each SOAC when the workload remains constant but is distributed differently across different amounts of blocks. A '-' means no result. All tests use 32-bit values.

Both tables show that the performance of **map** is only slightly improved from the original program, with only the test having an inner dimension of 128 showing some loss in performance. At this inner dimension size, the group size of the generated kernels will have only 32 threads, which might be to blame for the slight performance hit. However, table 5 shows that even when going above the normal cap of 1024 on the inner dimension, it still keeps a high effective bandwidth, indicating that **map** is well suited for having these larger workloads. At a size of 4096, resulting in a group size of 1024 threads, it still outperforms the original version at only 1024 elements pr. block. The slight performance hit going from 1024 to 4096 is also to be expected as there is more work for each block to do. The **map** tests also tell us that the tiling code does not seem to slow down the program in any considerable way. It still reads the same amount but stages it in shared memory before starting the actual computation.

reduce benefits from considerable speedup following the optimization. Keeping a constant inner dimension of 1024 and only increasing the number of blocks, the speedup of the optimized version sits solidly around a factor of 2.5. **reduce** was expected to be the operator to gain the most from the transformation, since, besides benefitting from the general properties of sequentialization, the size of the reduction tree to be done in shared memory is also shrunk from $\log n$ to $\log(n/q)$, where q is the sequentialization factor, as

outlined in section 7.2.3. There is however a steady decline in the effective bandwidth when increasing the amount of blocks which is not the case for the original program. One factor here might be the occupancy of the GPU in the transformed program. After transformation, all blocks require more resources in the form of registers and shared memory, meaning that a single SM might not be able to support the same amount of blocks as it could for the original program. Further, some resources might go to waste if the required resources for all the blocks an SM can support do not take up all available resources. As such, having more blocks will decrease the effective bandwidth if fewer blocks are running in parallel but still have more blocks to process. Furthermore, the second test shows that the effective bandwidth of **`reduce`** peaks when the inner dimension is 256 but the size at which it is the fastest compared to the original version is 1024. We also see the bandwidth when going beyond the original cap of 1024 where it is still quite good compared to the original program within the cap.

`scan` also gets a considerable speedup after the optimization, sitting at a factor of $\approx 1.9 - 2$ for all tests, comparable to what was found in the initial experiments with CUDA **`scan`** in section 4.1. **`scan`** was also expected to benefit from the optimizations for the same reasons as for **`reduce`** albeit not as much as it has to perform more intermediate steps. This can be seen, as the same trends appear for **`scan`** as they did for **`reduce`** but with a slightly lower bandwidth overall. From the first test, we see that we again have a decline in bandwidth, and also slightly in speedup, when increasing the amount of blocks to be processed. Again, this is believed to be an occupancy issue, but for **`scan`** this might be an even greater issue as it uses both more shared memory and more registers. Also, **`scan`** just has more intermediate steps than **`reduce`**, which will also factor into the lower performance gains. From the second test, it seems there is a steep performance drop when going beyond the original cap of 1024. That said, at those sizes, the bandwidth is still an improvement from the ones of the original program within the cap.

8.3 Scatter

The same two test setups were used for **`scatter`** as for the other SOACs with a few minor modifications. First, in the first test set-up, the number of blocks has been decreased to not exceed the amount of available memory on the system, since the **`scatter`** kernel consumes considerably more than the others as it reads in 2-3 matrices of size `[n][m]`, since a single **`scatter`** operation needs an array to update, an array of which indices to update, and an array with the values to be written. These will be staged in shared memory whereas the other SOACs simply read in a single matrix of size `[n][m]` on which they should operate. For this same reason, tests were not run on matrices with an inner size larger than 2048. Secondly, for each setup, two different test programs were used for **`scatter`** with the only difference being how the indices to update are calculated. One version reads in a randomly generated matrix of indices of size `[n][m]`. The other version uses an **`iota`** operator to generate indices from $0, \ldots, M - 1$ such that each element of each block is updated. Ultimately, both versions update the same number of indices with one version using completely random indices and the other consecutive in-bounds indices.

The results for the **`scatter`** benchmarks can be seen below in tables 6 and 7.

Number of blocks		50k	100k	150k	200k	250k
scatter random	org(GB/s)	1666.46	1662.51	1670.88	1670.87	1672.25
	seq(GB/s)	1062.77	1071.47	1072.38	1069.39	1073.10
	speedup	0.6377	0.6444	0.6418	0.6400	0.6417
scatter iota	org(GB/s)	1187.97	1200.16	1208.19	1213.12	1214.17
	seq(GB/s)	1339.32	1353.96	1359.57	1357.55	1361.41
	speedup	1.1274	1.1279	1.1252	1.1190	1.1212

Table 6: The effective bandwidth in GB/s and speedup for **scatter** with 1024 threads pr. block. The *org* row is the bandwidth before optimization and the *seq* row after. *Random* is with updates to random indices, *iota* updates all indices from $0, \ldots, M-1$ where M is the inner size of the matrix. All tests are with 32-bit values, 64-bit indices.

Sizes		[1048576] [128]u32	[524288] [256]u32	[262144] [512]u32	[131072] [1024]u32	[65536] [2046]u32
scatter random	org (GB/s)	1601.55	1680.91	1677.28	1670.64	-
	seq (GB/s)	1285.23	1063.17	1070.63	1071.78	1094.43
	speedup	0.7668	0.6324	0.6383	0.6415	-
scatter iota	org (GB/s)	1294.57	1289.86	1273.00	1197.95	-
	seq (GB/s)	1293.59	1361.43	1362.03	1358.43	1342.62
	speedup	0.9992	1.0554	1.0699	1.1339	-

Table 7: The effective bandwidth and speedup of **scatter** when the workload remains constant but is distributed differently across different amounts of blocks. '-' is no result. *Random* is with updates to random indices, *iota* updates all indices from $0, \ldots, M-1$ where M is the inner size of the matrix. All tests use 32-bit values, 64-bit indices.

For **scatter** there is a large difference between the version using random indices and the iota version using consecutive indices. The iota version generally shows good results with a slight speedup and performs best with a block size of 1024 where the speedup is $\approx 13\%$. This is close to the results achieved with **map** which is to be expected as **map** and **scatter** are similar in the sense that the transformation does not add any intermediate steps, but simply incorporates a loop into each thread to have them process several elements consecutively.

Interestingly, for the tests using random indices, the optimized version is significantly slower than the original one. This slowdown is not because the optimized version is particularly slow but because the original one has an incredibly high effective bandwidth. Exactly why this is the case for the original program is not entirely clear. One reason could be that the original program starts by reading in the indices and if it can determine that all or some of these are out of bounds, then it does not need to load and write the values, meaning less access to global memory. The optimized version will however always read a tile of all three input arrays, meaning more access to global memory than the original. For this reason, the iota tests are probably more indicative of the actual performance gains as you would assume that most, if not all, of the indices are valid.

8.4 Big Number Addition

Big number addition is a practical example of a program where the transformation is expected to have a significant impact. The implementation consists only of an outer **map**, with a nested **map**, **scan** and **map**, matching perfectly the two-level parallelism of the hardware. For this program, we have a single test where the workload is kept constant and distributed across a different number of blocks and group sizes like for the SOAC tests. The results can be seen in table 8. Note that only the transformed program is run with an inner size of 2048, as this would be too great for the original version. The reason for not having tests with sizes greater than 2048, is because the transformed program begins to encounter shared memory shortage. It is also worth noting that this program takes two two-dimensional arrays as input, so the size of the array for each of the test cases is read twice, but only written once. This also means that the transformed program will tile both input arrays, further increasing the impact on the amount of shared memory used.

Sizes	[1048576][128]u32	[524288][256]u32	[262144][512]u32	[131072][1024]u32	[65536][2048]u32
org (GB/s)	458.80	457.96	452.62	422.18	-
seq (GB/s)	805.90	824.04	841.60	782.81	677.00
speedup	1.7565	1.7993	1.8593	1.8541	-

Table 8: The results of our transformation on the big number addition program when distributing the workload differently across a number of blocks and block sizes.

The results show a good speedup ranging from 75% to 85% based on the size of the inner dimension. The speedup peaks with an inner size of 512 which is in line with results observed from the tests of **scan** in table 5 which is the dominating SOAC used in the computation. As we also saw, the inner **maps** do not affect the overall speedup a great deal. Finally, we have the test where the inner dimension is above the previous cap of 1024 where we see a fairly large drop off in the effective bandwidth. Again, this is in line with the results observed for **scan** but might be further amplified by the use of additional registers needed by the **maps**.

8.5 Intra-block Radix Sort

Finally, benchmarks were run on an intra-block version of radix sort, implemented using `partition2` [16]. It was decided to run on an intra-block version as opposed to doing a full grid-wide sort as in the CUDA version 4.2, to focus on demonstrating the results of the optimization. `partition2` takes a predicate and an array and returns an array of the same size where all the elements that succeed in the predicate come first, followed by all that fail. Using a bit split as in the CUDA version of radix sort, this can be used for general sorting. This way of implementing radix sort is a prime candidate for efficient sequentialization, as `partition2` can be implemented through the use of **map** , **scan** , and **scatter** . The results can be seen below in table 9.

Before discussing the results of the test there is one thing to note. Futharks own testing framework can compute the amount of bytes read and written to global memory from a specific kernel. However, we found that it thought 50% more bytes was transferred than was the case, meaning it thought that some data had to either be read twice or written twice. Examining the generated CUDA code, showed that only two accesses to global memory are present. One at the beginning to load the tile to shared memory and one at the end

Sizes	[1048576][128]u32	[524288][256]u32	[262144][512]u32	[131072][1024]u32
org (GB/s)	70.09	69.09	64.68	54.22
seq (GB/s)	94.68	107.05	112.91	112.29
speedup	1.3352	1.5493	1.7619	2.0693

Table 9: The results of our transformation on the intra-group radix program when distributing the workload differently across a number of blocks and clock sizes.

to write, the now sorted tile, back to global memory. Further, it only listed the amount of bytes transferred for a single invocation of the kernel, but here the kernel is wrapped in a loop with 8 iterations so we have scaled the results accordingly.

The results show that the program has the largest speedup when the inner dimension size is 1024. With the optimization applied this results in a kernel with a block size of 256 threads, and it is also at this inner size the original program performs well, with only a size of 128 performing slightly better, but this might just be noise. So with optimization or not, the program seems to favor a block size of around 256 threads.

9 Discussion

9.1 Results

Overall the results demonstrate how programs that exhibit intra-group parallelism can benefit considerably from the efficient sequentialization optimization. **`scan`** and **`reduce`** are two of the most commonly used parallel operators in this style of data-parallel programming and both benefit from considerable speed-up. Furthermore, while not as prominently, both **`map`** and **`scatter`** using consecutive indices, also benefit from a speed-up. Secondly, the optimization allows for even larger inner sizes as each thread can now process multiple elements leading to a smaller number of total blocks needed for the kernel. Larger inner sizes like `2048` and `4096` are not possible without sequentialization as the CUDA programming model caps the maximum number of threads per block at 1024. Having a larger range of inner sizes available to the programmer allows a larger degree of freedom in tailoring the kernel parameters specifically to the given program.

However, the benchmarks have also shown a couple of downsides to the implementation. First, applying sequentialization results in kernels that use considerably more shared memory. In itself, this is not an issue, but it is one thing to be wary of when running programs with large input sizes as it can cap the inner sizes as seen in the results. Furthermore, applying the optimization to programs dominated by the use of **`scatter`** to write to non-consecutive indices should be done with caution and accompanied by tests that investigate the execution time before and after applying the optimization, as random indices have been shown to potentially cause some slow-down.

Normally a Futhark program will contain multiple different code versions utilizing different amounts of the nested parallelism available. In this project, we have forced it to only produce the one that results in an intra-group kernel, as this was the target for our transformation. This allowed for easy testing of different inputs on the original and optimized versions. However, Futhark might normally have decided for some input that one of the other produced code version were a better fit, but since it only now had the intra-group one that was chosen instead. Therefore, while our results show overall good results compared to the intra-group kernel, it might not be faster than the one Futhark would normally have chosen. Further, the addition of our optimization should likely have an effect on the threshold used when Futhark discriminates which version to choose. Right now the threshold is based on if the data fits within an intra-group kernel, but as seen we can now go up to higher inner sizes than before, but this is not reflected in the threshold.

We suspected that some of the performance losses when increasing the inner size were due to occupancy issues. There is however no explicit way of knowing exactly how much shared memory or how many registers the kernels we generate are going to use. For this, we imagine you would have to implement the optimization at the GPUmem representation rather than just GPU and keep track of how much-shared memory is allocated and how many registers are used. This however introduces the question of what should happen if we at some point reach the maximum amount of resources used. Should any further encountered arrays just not be tiled or the chunks not be loaded to registers? The best solution would probably be some form of analysis to see which arrays have the most reads and writes and move those to shared memory, while once that are rarely used could be kept in global memory. This is however far outside the scope of this project.

9.2 Future Work

While our results show that the project has been an overall success, there are still areas where additional performance might be possible. Additionally, there are still some cases that the current implementation is unable to handle. As such suggestions for future work are;

Tiling: The code generated for tiling some piece of an array is the same no matter the base primitive type of the array. This means that whether they are 1-byte or 8-byte elements they are all read one by one. For small-size elements, it would however be possible to read multiple values at once. If the smallest size the hardware can read is 4 bytes and the primitive type of the array is 1 byte in size, then all 4 elements could be transferred in one go by a single thread. This would have the additional benefit of reducing the amount of bank conflicts.

`SegHist`: The compiler contains an additional `SegOp` called `SegHist` which is not currently handled by our implementation. It was not needed to examine this operator in detail to see that it was not needed to implement the transformations for the SOACs, but it should still be handled in some way to make the implementation complete.

Multiple dimensions: The current implementation falls back to the original program if an array of more than two dimensions is encountered. The reason for this is, that it is not clear at which dimension efficient sequentialization should take place and as discussed simply flattening the inner dimensions does not seem to be the solution. A more thorough analysis of what should happen in such a case is needed after which the implementation could be extended to handle it.

Registers: There might be more opportunity to keep values in registers rather than staging them in shared memory between operations. When applying the transformation on e.g. a **`segscan`** multiple new segmented operators are created, between which data could potentially be kept in registers. Even between multiple already existing segmented operators at the thread level, it might be possible to see if the data needs to be staged or if the same threads will work with the results they produced for the previous operator.

Better analysis: When the transformation is successfully applied it produces code that discriminates between the sequentialized version and the original version. The way this is currently done is by having a fixed threshold T and comparing that to the number of threads the optimized version would use. If the number of threads is greater than or equal to T, then our version is run otherwise the original one is run. Not much time was spent examining what this threshold should be as both code versions should be put in line with the rest of the code versions generated by incremental flattening.

Representation: Implementing the transformation at the GPU representation level inside the compiler means we do not have any explicit memory control. As such it is very difficult to estimate the amount of shared memory or registers that the program is going to use when compiled. One could consider moving the optimization to the GPUmem representation to have better control over this, which might allow for even more optimization as then it would be possible to explicitly reuse shared memory buffers, reducing the resource usage of each block.

Investigating performance of `scatter` : More experiments should be conducted to clarify when applying the optimization to a **`scatter`** operation results in speedup and when and why it sometimes results in slow-down.

10 Conclusion

This thesis has shown how efficient sequentialization can be implemented as a code transformation technique by implementing it as an optimization targeting intra-group kernels in the Futhark programming language. Specifically, transformations for the SOACs **`map`**, **`reduce`**, **`scan`** and **`scatter`** have been shown, achieving significant speedups while allowing the intra-group code versions of Futhark programs to work on larger sizes than they were able to before and at higher bandwidths. The compiler has further been extended to allow for a `seq_factor` attribute in the source language, allowing the end user to control the amount of sequentialization to be performed and enabling the optimization in the first place.

We described how the transformations rest on the mathematical foundation of list homomorphisms, which in itself enables the program to take better advantage of the hardware by using shared memory and reducing the number of intermediate buffers needed. However, this also causes the transformed programs to consume considerably more shared memory which is a limited resource and should be taken into account at large input sizes.

Finally, suggestions for future work have been laid out to fully integrate the optimization into the main Futhark compiler, such as modifying the thresholds for determining which code version should be run in order to reflect the new capabilities of the intra-group kernels.

References

[1] ADINETS, A. A faster radix sort implementation. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf. Accessed: 16.12.2023.

[2] BEN-NUN, T., DE FINE LICHT, J., ZIOGAS, A. N., SCHNEIDER, T., AND HOEFLER, T. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2019), SC '19, Association for Computing Machinery.

[3] BIRD, R. S. An introduction to the theory of lists. Tech. Rep. PRG56, OUCL, October 1986.

[4] BLELLOCH, G. E. Programming parallel algorithms. *Commun. ACM 39*, 3 (mar 1996), 85–97.

[5] DOCS, F. Futhark user's guide: futhark-test. https://futhark.readthedocs.io/en/latest/man/futhark-test.html. Accessed: 10.12.2023.

[6] FUTHARK-LANG. Futhark library documentation. https://futhark-lang.org/docs/prelude/. Accessed: 10.12.2023.

[7] GORLATCH, S. Systematic extraction and implementation of divide-and-conquer parallelism. In *Programming Languages: Implementations, Logics, and Programs* (Berlin, Heidelberg, 1996), H. Kuchen and S. Doaitse Swierstra, Eds., Springer Berlin Heidelberg, pp. 274–288.

[8] HASKELL.ORG. The haskell cabal | overview. https://www.haskell.org/cabal. Accessed: 13.12.2023.

[9] HENRIKSEN, T., AND OANCEA, C. E. A t2 graph-reduction approach to fusion. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing* (New York, NY, USA, 2013), FHPC '13, ACM, pp. 47–58.

[10] HENRIKSEN, T., SERUP, N. G. W., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. E. 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* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 556–571.

[11] HENRIKSEN, T., THORØE, F., ELSMAN, M., AND OANCEA, C. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2019), PPoPP '19, ACM, pp. 53–67.

[12] MOSES, W. S., CHURAVY, V., PAEHLER, L., HÜCKELHEIM, J., NARAYANAN, S. H. K., SCHANEN, M., AND DOERFERT, J. Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2021), SC '21, Association for Computing Machinery.

[13] MOSES, W. S., IVANOV, I. R., DOMKE, J., ENDO, T., DOERFERT, J., AND ZINENKO, O. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. In *Proceed-*

ings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (New York, NY, USA, 2023), PPoPP '23, Association for Computing Machinery, p. 119–134.

[14] NVIDIA. Cuda c++ programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Accessed: 16.12.2023.

[15] NVIDIA. Nvidia a100 tensor core gpu. `https://www.nvidia.com/en-us/data-center/a100/`. Accessed: 13.12.2023.

[16] OANCEA, C. E. Lecture notes for the software track of the pmph course. PDF, 2018. University of Copenhagen.

[17] OANCEA, C. E., AND HENRIKSEN, T. On the cuda implementation of reduce and scan. PDF, 2022. University of Copenhagen.

[18] SATISH, N., HARRIS, M., AND GARLAND, M. Designing efficient sorting algorithms for manycore gpus. *Proc. 23rd IEEE International Parallel and Distributed Processing Symposium* (2009).

[19] TROELS, H. Futhark: An optimising compiler for a functional, array-oriented language. `https://hackage.haskell.org/package/futhark`. Accessed: 14.11.2023.

A Transformation Rules

This appendix describes the notation originally used for the transformation rules. They are more closely based on the concepts from [11], which describes the *incremental flattening* algorithm used in Futhark to generate multiple code versions utilizing different amounts of parallelism. This notation was however found to be far more complex than needed for the transformation rules of the report, so we opted for the more simplified notation that closely resembles the actual intermediate representation of the compiler. These rules are simply here since they were already developed.

A.1 Notation

In the following sections we will use an extended version of two languages described in [11] as a means to create and discuss the transformation rules for the different operators. These two languages are particularly suitable as the first one, called the *source language* is morally a subset of Futhark, and the second, called the *target language*, is morally equivalent to the intermediate representation that the Futhark compiler works with internally [11].

In the languages x, y, and z are used to range over a denumerable infinite set of variables, d over integers, p over variables or integers, b over bools, and $\overline{q}$ to denote a sequence of objects of some kind. The context in which $\overline{q}$ is used denotes whether the separating character is a comma or simply a white space [11].

We will extend the languages with two new operators and a new expression. The first operator is **`scatter`** , with the same type and semantics as in section 3.1.2. The second is the $\texttt{split}_p$ operator with type as shown below. The intuition is that it splits some array into p chunks of equal size. The extended syntax is shown in figure 7.

$$\texttt{split}_p : [n]\alpha \to [n/p][p]\alpha$$

$$
\begin{array}{lll}
bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{>} \mid \cdots \\
op & ::= & \texttt{transpose} \mid \texttt{rearrange}\ (d, \ldots, d) \mid \texttt{replicate} \mid \texttt{split}_p \mid \texttt{scatter} \\
soac & ::= & \texttt{map} \mid \texttt{scan} \mid \texttt{reduce} \mid \texttt{redomap} \mid \texttt{scanomap} \\
e & ::= & x \mid d \mid b \mid (e, \ldots, e) \mid e[e] \mid e\ bop\ e \mid op\ e \cdots e \\
 & \mid & \textbf{loop}\ x_1 \cdots x_n = e\ \textbf{for}\ y < e\ \textbf{do}\ e \\
 & \mid & \textbf{let}\ x_1 \cdots x_n = e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
 & \mid & soac\ f\ e \cdots e \mid x\ \texttt{with}\ [y_1 \cdots y_n] = z_1 \cdots z_n \\
f & ::= & \lambda x_1 \cdots x_n \to e \mid soac\ f\ e \cdots e \mid e\ bop \mid bop\ e
\end{array}
$$

Figure 7: Extended syntax of the source language [11]

We also have **`redomap`** and **`scanomap`** as the composition of **`map`** /**`reduce`** and **`map`** /**`scan`** respectively. These are defined as follows. Note how **`redomap`** is the same composition used in the optimized map re-

duce lemma.

$$\begin{aligned}\texttt{scanomap}\ \odot\ f\ \overline{d}\ \overline{xs} &\equiv \texttt{scan}\ \odot\ \overline{d}\ (\texttt{map}\ \ f\ \overline{xs})\\ \texttt{redomap}\ \odot\ f\ \overline{d}\ \overline{xs} &\equiv \texttt{reduce}\ \odot\ \overline{d}\ (\texttt{map}\ \ f\ \overline{xs})\end{aligned}$$

Like in [11] the target language has the same syntax as the source language with added segmented operators of **segmap segred** and **segscan** and the modified semantics where the normal SOACs, **map** , **scan** and **reduce** are now sequential. However, We extend the notation for segmented SOACs such that we also denote the number of workers as a subscript to the operator. That is in $\texttt{segmap}^l_n\ \Sigma\ e$, n denotes the number of workers for that segmented operation. In OpenCL terms with $l = 1$, n would denote the group size and if $l = 0$, n would denote the number of threads. The extended syntax for the target language is reproduced in figure 8

$$\begin{array}{lcl} e & ::= & \cdots \\ & | & \texttt{segmap}^l_n\ \ \Sigma\ e \mid \texttt{segred}^l_n\ \ \Sigma\ f\ \overline{d}\ e \mid \texttt{segscan}^l_n\ \ \Sigma\ f\ \overline{d}\ e \\ \Sigma & ::= & \bullet \mid \Sigma, \langle \overline{x} \in \overline{y} \rangle \end{array}$$

Figure 8: Extended syntax for the target language [11]

Here Σ is the mapnest context of which the domain, $Dom(\Sigma)$ where $\Sigma = \Sigma', \langle \overline{x} \in \overline{y} \rangle$ is defined inductively as the set $\{\overline{x}\} \cup Dom(\Sigma')$, where $Dom(\bullet) = \emptyset$ as the base case [11].

The parallel segmented operators corresponds to a *perfect parallel nest* in which the innermost parallel construct is a **map** , **scanomap** or **redomap** for a **segmap** , **segscan** and **segred** respectively [11]. That is. if we have $\Sigma = \langle \overline{x_p} \in \overline{y_p} \rangle, \cdots, \langle \overline{x_1} \in \overline{y_1} \rangle$ for some $p \geq 1$ then we have

$$\begin{aligned}\texttt{segmap}^l_n\ \ \Sigma\ e &\equiv \texttt{map}\ (\lambda \overline{x_p} \to \texttt{map}\ (\lambda \overline{x_{p-1}} \to \cdots \texttt{map}\ (\lambda \overline{x_1} \to e) \overline{y_1}) \overline{y_{p-1}} \cdots)\ \overline{y_p}\\ \texttt{segred}^l_n\ \ \Sigma\ \odot\ \overline{d}\ e &\equiv \texttt{map}\ (\lambda \overline{x_p} \to \texttt{map}\ (\lambda \overline{x_{p-1}} \to \cdots \texttt{redomap}\ \ \odot (\lambda \overline{x_1} \to e) \overline{d}\ \overline{y_1}) \overline{y_{p-1}} \cdots)\ \overline{y_p}\\ \texttt{segscan}^l_n\ \ \Sigma\ \odot\ \overline{d}\ e &\equiv \texttt{map}\ (\lambda \overline{x_p} \to \texttt{map}\ (\lambda \overline{x_{p-1}} \to \cdots \texttt{scanomap}\ \ \odot (\lambda \overline{x_1} \to e) \overline{d}\ \overline{y_1}) \overline{y_{p-1}} \cdots)\ \overline{y_p}\end{aligned}$$

A.2 Rules

This section will introduce the transformation rules for the three SOACs. They will be presented in both the source and target language described. Having the rules in the source language gives the best intuition on what the transformation is supposed to do, but the rules in the target language are more detailed and closer to the intermediate representation in the actual Futhark compiler and can thus help guide the implementation. Explaining the rules in these higher-level languages also means we can focus on the semantics and do not need to concern ourselves with where arrays are allocated on the GPU. Following each rule we will also argue that the semantics of the programs stay the same and that the work and depth complexity are not affected.

A.2.1 Map

```
map (λxs → map f xs) xss
≡
map (λxs →
  map (λx →
    map f x
  ) (split_{m/q} xs)
) xss
```

Listing 31: High level map transformation rule.

```
segmap^1_n ⟨xs ∈ xss⟩ (
  segmap^0_m ⟨x ∈ xs⟩ e
)
≡
segmap^1_n ⟨xs ∈ xss⟩ (
  let m' = ⌈m/q⌉
  in segmap^0_{m'} ⟨x ∈ split_{m'} xs⟩ (
    map (λy → e) x
  )
)
```

Listing 32: Low level map transformation rule.

The high-level and low-level transformation rules for map can be seen in listing 8 and 9 respectively, where `xss` has two dimensions. Looking at the high-level rule does not convey much new information except that `xs` is further split up into chunks which are then mapped over. However, in the low-level rule, we see that the two outer maps are segmented maps at two different levels of hardware parallelism with the innermost being a sequential map, now working on the chunk of `xs`. Further the outermost **segmap** has had its number of workers reduced by the sequentialization factor ϵ.

A.2.2 Reduce

```
map (λxs → reduce ⊕ 0_⊕ xs) xss
≡
map (λxs →
  let ys = map (λx →
    reduce ⊕ 0_⊕ x
  ) (split_{m/q} xs)
  in reduce ⊕ 0_⊕ ys
) xss
```

Listing 33: High level reduce transformation rule.

```
segmap^1_n ⟨xs ∈ xss⟩ (
  segred^0_m ⟨x ∈ xs⟩ ⊕ d e
)
≡
segmap^1_n ⟨xs ∈ xss⟩ (
  let m' = ⌈m/q⌉
  let ys =
    segmap^0_{m'} ⟨x ∈ split_{m'} xs⟩ (
      redomap ⊕ (λy → e) d x
    )
  in segred^0_{m'} ⟨y ∈ ys⟩ ⊕ d y
)
```

Listing 34: Low level reduce transformation rule.

The two transformation rules for reduce can be seen in figure 33 and 34. In the high-level rule we manifest a new map on top of the reduction, which now reduces over a chunk of the original xs array. That gives a reduction result for each chunk, which is then reduced over with the same operator and neutral element to get the final result.

The second rule for the target language is a bit more involved. The same intuition exists in that the original

x is split up into chunks which are then reduced over, sequentially, in the **redomap** and the result of this is then reduced over in parallel in the final **segred**. However, to see why the original expression e ends up in the **redomap** and the expression in the final **segred** is simply y we will use the rewrite rules for **segred** along with the optimized map-reduce lemma.

$$\begin{aligned}
\textbf{segred}\ \langle x \in xs \rangle \oplus d\ e &\equiv \textbf{redomap}\ \oplus\ (\lambda x \to e)\ d\ xs \\
&\equiv (\textbf{reduce}\ \oplus\ d) \circ (\textbf{map}\ (\lambda x \to e))\ xs \\
&\equiv (\textbf{reduce}\ \oplus\ d) \circ (\textbf{map}\ (\textbf{redomap}\ \oplus\ (\lambda x \to e)\ d)) \circ \textbf{split}_{m/q}\ xs \\
&\equiv \textbf{segred}\ \langle x \in \textbf{split}_{m/q} xs > \oplus\ d\ (\textbf{redomap}\ \oplus\ (\lambda x \to e)\ d)
\end{aligned}$$

which is the first part of the transformed program.

Secondly we have that;

$$\begin{aligned}
\textbf{reduce}\ \oplus\ \overline{d}\ \ \overline{ys} &\equiv \textbf{redomap}\ \oplus (\lambda y \to y)\ \overline{d}\ \overline{ys} \\
&\equiv \textbf{segred}\ \langle y \in \overline{ys} \rangle \oplus\ \overline{d}\ y
\end{aligned}$$

which is the second part of the transformed program

A.2.3 Scan

```
map (λxs → scan ⊕ 0⊕ xs) xss
≡
map (λxs →
  let ys = map (λx →
    reduce ⊕ 0⊕ x
  ) (split_ε xs)
  let zs = scan ⊕ 0⊕ ys
  in map (λx z →
    scan ⊕ z x
  ) (split_ε xs) zs
) xss
```

Listing 35: High level scan transformation rule.

```
segmap^1_n ⟨xs ∈ xss⟩ (
    segscan^0_m ⟨x ∈ xs⟩ ⊕ d e
)
≡
segmap^1_n ⟨xs ∈ xss⟩ (
    let m' = ⌈m/q⌉
    let ys =
        segmap^0_m' ⟨x ∈ split_q xs⟩ (
            redomap ⊕ (λy → e) d x
        )
    let zs =
        segscan_incl^0_m' ⟨y ∈ ys⟩ ⊕ d y
    in segmap^0_m' ⟨z ∈ zs⟩⟨x ∈ split_q xs⟩ (
        scanomap ⊕ (λy → e) d x
    )
)
```

Listing 36: Low level scan transformation rule.

The transformation rules can be seen in figure 35 and 36. To successfully split the scan up between multiple chunks, each chunk needs to know the final sum of the previous chunks. The, pr. chunk sums are first

computed with the reduce, which is then scanned across, to get the sum of previous chunks concerning all the chunks. Then these, cross-chunk scan results, can then be used to perform the final scan.

A.2.4 Scatter

```
map (λ ds is vs →
    scatter ds is vs
) dss iss vss
≡
map (λ ds is vs →
    loop ds' = ds
    for j < q do
        map (λ i v →
            scatter ds i v
        ) (split_{m/q} is) (split_{m/q} vs)
) dss iss vss
```

Listing 37: High level scatter transformation rule.

```
segmap^1_n ⟨ds ∈ dss⟩⟨is ∈ iss⟩⟨vs ∈ vss⟩ (
    segmap^0_m ⟨i ∈ is⟩⟨v ∈ vs⟩ (
        ds with [i] = v
    )
)
≡
segmap^1_n ⟨ds ∈ dss⟩⟨is ∈ iss⟩⟨vs ∈ vss⟩ (
    let m' = ⌈m/q⌉
    loop ds' = ds
    for j < q do
        segmap^0_{m'} ⟨i ∈ split_{m'} is⟩
                      ⟨v ∈ split_{m'} vs⟩ (
            ds' with [i] = v
        )
)
```

Listing 38: Low level scatter transformation rule.

Neither notation lends itself particularly well to describe the transformation for **scatter**. The general idea is that **scatter** distributes the indices from is and values from vs into chunks for each thread to process. The reason the destination ds is not chunked is that a thread might need to scatter elements to indices that would be out of the bounds of the said chunk. All threads have access to all of ds. There is no actual segmented scatter operator in the intermediate representation. Instead, a **scatter** is represented with a **segmap** which does not return anything but rather performs an in-place update of some array.

B IntraSeq Module

```
1  {-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
2  {-# HLINT ignore "Use zipWith" #-}
3  {-# HLINT ignore "Use uncurry" #-}
4  {-# OPTIONS_GHC -Wno-name-shadowing #-}
5  {-# HLINT ignore "Use lambda-case" #-}
6  {-# HLINT ignore "Replace case with maybe" #-}
7  {-# LANGUAGE TypeFamilies #-}
8  {-# OPTIONS_GHC -Wno-orphans #-}
9  module Futhark.Optimise.IntraSeq (intraSeq) where
10
11 import Language.Futhark.Core
12 import Futhark.Pass
13 import Futhark.IR.GPU
14 import Futhark.Builder.Class
15 import Futhark.Construct
16 import Futhark.Transform.Rename
17
18 import Control.Monad.Reader
19 import Control.Monad.State
20
21 import Data.Map as M
22 import Data.IntMap.Strict as IM
23 import Data.List as L
24 import Data.Set as S
25
26 import Debug.Pretty.Simple
27 import Debug.Trace
28 import Data.Sequence
29 import Control.Monad.Trans.Except
30 import Control.Monad.Except
31 import Futhark.Transform.Substitute
32 import Futhark.IR.GPU.Simplify (simplifyGPU)
33
34 type SeqM a = ReaderT (Scope GPU) (State VNameSource) a
35
36 -- | A builder with additional fail functionality
37 type SeqBuilder a = ExceptT () (Builder GPU) a
38
39 runSeqBuilder ::
40   (MonadFreshNames m, HasScope GPU m, SameScope GPU GPU) =>
```

```
41    SeqBuilder a ->
42    m (Maybe (Stms GPU))
43  runSeqBuilder (ExceptT b) = do
44    (tmp, stms) <- runBuilder b
45    case tmp of
46      Left _ -> pure Nothing
47      Right _-> pure . Just $ stms
48
49
50  collectSeqBuilder ::
51    SeqBuilder a ->
52    SeqBuilder (a, Stms GPU)
53  collectSeqBuilder (ExceptT b) = do
54    (tmp, stms) <- lift $ do collectStms b
55    case tmp of
56      Left _ -> throwError ()
57      Right x -> pure (x, stms)
58
59  collectSeqBuilder' ::
60    SeqBuilder a ->
61    SeqBuilder (Stms GPU)
62  collectSeqBuilder' (ExceptT b) = do
63    (tmp, stms) <- lift $ do collectStms b
64    case tmp of
65      Left _ -> throwError ()
66      Right _ -> pure stms
67
68
69
70
71  runSeqMExtendedScope :: SeqM a -> Scope GPU -> Builder GPU a
72  runSeqMExtendedScope m sc = do
73    scp <- askScope
74    let sc' = sc <> scp
75    let tmp = runReaderT m sc'
76    st <- get
77    let tmp' = runState tmp st
78    pure $ fst tmp'
79
80
81  -- | A structure for convenient passing of different information needed
       at
```

```haskell
-- various stages during the pass.
data Env = Env {
  grpId      :: SubExp,             -- The group id
  grpSize    :: SubExp,             -- The group size after seq
  grpsizeOld :: SubExp,             -- The group size before seq
  threadId   :: Maybe VName,        -- the thread id if available
  nameMap    :: M.Map VName VName,  -- Mapping from arrays to tiles
  seqFactor  :: SubExp
}
  deriving(Show)

setMapping :: Env -> M.Map VName VName -> Env
setMapping (Env gid gSize gSizeOld tid _ factor) mapping =
            Env gid gSize gSizeOld tid mapping factor

updateMapping :: Env -> M.Map VName VName -> Env
updateMapping env mapping =
  let mapping' = mapping `M.union` nameMap env
  in setMapping env mapping'

memberMapping :: Env -> VName -> Bool
memberMapping env name = M.member name (nameMap env)

lookupMapping :: Env -> VName -> Maybe VName
lookupMapping env name
  | M.member name (nameMap env) = do
    case M.lookup name (nameMap env) of
      Just n ->
        case lookupMapping env n of
          Nothing -> Just n
          n' -> n'
      Nothing -> Nothing
lookupMapping _ _ = Nothing

updateEnvTid :: Env -> VName -> Env
updateEnvTid (Env gid sz szo _ tm sq) tid = Env gid sz szo (Just tid)
    tm sq

getThreadId :: Env -> VName
getThreadId env =
  case threadId env of
    (Just tid ) -> tid
```

```
123     _ -> error "No tid to get"
124
125 findSeqAttr :: Attrs -> Maybe Attr
126 findSeqAttr (Attrs attrs) =
127   let attrs' = S.toList attrs
128   in case L.findIndex isSeqFactor attrs' of
129     Just i -> Just $ attrs' !! i
130     Nothing -> Nothing
131   where
132     isSeqFactor :: Attr -> Bool
133     isSeqFactor (AttrComp "seq_factor" [AttrInt _]) = True
134     isSeqFactor _ = False
135
136 getSeqFactor :: Attrs -> SubExp
137 getSeqFactor attrs =
138   case findSeqAttr attrs of
139     Just i' ->
140       let (AttrComp _ [AttrInt x]) = i'
141       in intConst Int64 x
142     Nothing -> intConst Int64 4
143
144 shouldSequentialize :: Attrs -> Bool
145 shouldSequentialize attrs =
146   case findSeqAttr attrs of
147     Just _ -> True
148     Nothing -> False
149
150
151 intraSeq :: Pass GPU GPU
152 intraSeq =
153     Pass "name" "description" $
154       intraproceduralTransformation onStms
155         >=> simplifyGPU
156     where
157       onStms scope stms =
158         modifyNameSource $
159           runState $
160             runReaderT (seqStms stms) scope
161
162 -- SeqStms is only to be used for top level statements. To
    sequentialize
163 -- statements within a body use seqStms'
```

```
164 seqStms ::
165   Stms GPU ->
166   SeqM (Stms GPU)
167 seqStms stms = do
168   tmp <- runSeqBuilder $ forM (stmsToList stms) seqStm
169   case tmp of
170     Nothing -> pure stms
171     Just stms' -> pure stms'
172
173 -- | Matches against singular statements at the group level. That is
      statements
174 -- that are either SegOps at group level or intermediate statements
      between
175 -- such statements
176 seqStm ::
177   Stm GPU ->
178   SeqBuilder ()
179 seqStm stm@(Let pat aux (Op (SegOp (
180               SegMap (SegGroup virt (Just grid)) space ts
181                      (KernelBody _ stms kres)))))
182   | L.length (unSegSpace space) /= 1 = lift $ do addStm stm
183   | not $ shouldSequentialize (stmAuxAttrs aux) = lift $ do addStm stm
184   | otherwise = do
185
186     let seqFactor = getSeqFactor $ stmAuxAttrs aux
187     let grpId   = fst $ head $ unSegSpace space
188     let sizeOld = unCount $ gridGroupSize grid
189     sizeNew <- lift $ do letSubExp "group_size" =<< eBinOp (SDivUp
      Int64 Unsafe)
190                                                    (eSubExp sizeOld)
191                                                    (eSubExp seqFactor)
192
193     let env = Env (Var grpId) sizeNew sizeOld Nothing mempty seqFactor
194
195     -- As we need to catch 'internal' errors we use runSeqBuilder here
196     res <- runSeqBuilder $ do
197
198
199           exp' <- buildSegMap' $ do
200               env' <- mkTiles env
201
```

```
202               let grid' = Just $ KernelGrid (gridNumGroups grid) (Count
    sizeNew)
203               let lvl' = SegGroup virt grid'
204
205               _ <- seqStms' env' stms
206
207               kres' <- lift $ do flattenResults pat kres
208               pure (kres', lvl', space, ts)
209
210           lift $ do addStm $ Let pat aux exp'
211
212     -- Based on error or not we now return the correct program
213     case res of
214       Nothing -> lift $ do addStm stm
215       -- Just stms' -> lift $ do addStms stms'
216       Just stms'
217         | not $ isOneStm (stmsToList stms') -> lift $ do addStm stm
218         | otherwise -> do
219             let [stm'] = stmsToList stms'
220
221             -- Create the braches with each code version
222             body1 <- lift $ do mkMatchBody stm'
223             body2 <- lift $ do mkMatchBody stm
224
225             -- Create the conditional statements
226             cond <- lift $ do eCmpOp (CmpSle Int64) (eSubExp $ intConst
    Int64 32) (eSubExp $ grpSize env)
227
228             matchExp <- lift $ do eIf' (pure cond) (pure body1) (pure
    body2) MatchEquiv
229
230             lift $ do addStm (Let pat aux matchExp)
231
232     where
233       isOneStm :: [Stm GPU] -> Bool
234       isOneStm [_] = True
235       isOneStm _   = False
236
237       mkMatchBody :: Stm GPU -> Builder GPU (Body GPU)
238       mkMatchBody stm = do
239           let (Let pat' aux' exp') = stm
240           newPat <- renamePat pat'
```

```
241           newExp <- renameExp exp'
242           let newStm = Let newPat aux' newExp
243           let (Let pat' _ _) = newStm
244           let pNames = L.map patElemName $ patElems pat'
245           let res = L.map (SubExpRes mempty . Var) pNames
246           pure $ Body mempty (stmsFromList [newStm]) res
247
248
249
250
251 seqStm (Let pat aux (Match scrutinee cases def dec)) = do
252   cases' <- forM cases seqCase
253   let (Body ddec dstms dres) = def
254   dstms' <- collectSeqBuilder' $ forM (stmsToList dstms) seqStm
255   (dres', stms') <- collectSeqBuilder $ localScope (scopeOf dstms') $
       fixReturnTypes pat dres
256   let def' = Body ddec (dstms' <> stms') dres'
257   lift $ do addStm $ Let pat aux (Match scrutinee cases' def' dec)
258   where
259     seqCase :: Case (Body GPU) -> SeqBuilder (Case (Body GPU))
260     seqCase (Case cpat body) = do
261       let (Body bdec bstms bres) = body
262       bstms' <- collectSeqBuilder' $
263                   forM (stmsToList bstms) seqStm
264       (bres', stms') <- collectSeqBuilder $ localScope (scopeOf bstms')
       $ fixReturnTypes pat bres
265       let body' = Body bdec (bstms' <> stms') bres'
266       pure $ Case cpat body'
267
268 seqStm (Let pat aux (Loop header form body)) = do
269   let fparams = L.map fst header
270   let (Body bdec bstms bres) = body
271   bstms' <- collectSeqBuilder' $
272               localScope (scopeOfFParams fparams) $
273                 forM_ (stmsToList bstms) seqStm
274   (bres', stms') <- collectSeqBuilder $
275                       localScope (scopeOf bstms') $
276                         fixReturnTypes pat bres
277   let body' = Body bdec (bstms' <> stms') bres'
278   lift $ do addStm $ Let pat aux (Loop header form body')
279
```

```
280 -- Catch all pattern. This will mainly just tell us if we encounter some
281 -- statement in a test program so that we know that we will have to handle it
282 seqStm stm = lift $ do addStm stm
283
284
285 -- | Much like seqStms but now carries an Env
286 seqStms' ::
287   Env ->
288   Stms GPU ->
289   SeqBuilder ()
290 seqStms' env stms = do forM_ (stmsToList stms) (seqStm' env)
291
292
293
294 -- |Expects to only match on statements at thread level. That is SegOps at
295 -- thread level or statements between such SegOps
296 seqStm' ::
297   Env ->
298   Stm GPU ->
299   SeqBuilder ()
300 seqStm' env (Let pat aux
301              (Op (SegOp (SegRed lvl@(SegThread {}) space binops ts kbody ))))
302   | L.length (unSegSpace space) /= 1 = throwError ()
303   | differentSize space env = throwError ()
304   | otherwise = do
305       let tid = fst $ head $ unSegSpace space
306       let env' = updateEnvTid env tid
307
308       -- thread local reduction
309       reds <- lift $ do mkIntmRed env' kbody ts binops
310       kbody' <- lift $ do mkResultKBody env' kbody reds
311
312       -- Update existing SegRed
313       -- we need numResConsumed because reduction result types are unchanged
314       let numResConsumed = numArgsConsumedBySegop binops
315       let space' = SegSpace (segFlat space) [(tid, grpSize env')]
316
```

```
317       tps <- mapM lookupType reds
318       let ts' = L.map (stripArray 1) tps
319       pat' <- updateSegOpPatTypes numResConsumed pat tps
320       lift $ do addStm $ Let pat' aux (Op (SegOp (SegRed lvl space'
    binops ts' kbody')))
321
322 seqStm' env stm@(Let pat _ (Op (SegOp
323           (SegMap lvl@(SegThread {}) space ts kbody))))
324   | L.length (unSegSpace space) /= 1 = throwError ()
325   | differentSize space env = throwError ()
326   | isScatter kbody = seqScatter env stm
327   | otherwise = do
328       let tid = fst $ head $ unSegSpace space
329       exp <- buildSegMap' $ do
330         phys <- newVName "phys_tid"
331         let env' = updateEnvTid env tid
332         usedArrays <- lift $ do getUsedArraysIn env kbody
333         iot <- lift $ do buildSeqFactorIota env
334         lambSOAC <- lift $ do buildSOACLambda env' usedArrays iot kbody
     ts
335         let screma = mapSOAC lambSOAC
336         chunks <- lift $ do mapM (getChunk env') usedArrays
337         res <- lift $ do letTupExp' "res" $ Op $ OtherOp $
338                          Screma (seqFactor env) (chunks ++ [iot])
    screma
339         let space' = SegSpace phys [(tid, grpSize env)]
340         let types' = scremaType (seqFactor env) screma
341         let kres = L.map (Returns ResultMaySimplify  mempty) res
342         pure (kres, lvl, space', types')
343
344       let names = patNames pat
345       lift $ do letBindNames names exp
346
347 seqStm' env (Let pat aux
348             (Op (SegOp (SegScan (SegThread {}) space binops ts kbody)))
    )
349   | L.length (unSegSpace space) /= 1 = throwError ()
350   | differentSize space env = throwError ()
351   | otherwise = do
352       usedArrays <- lift $ do getUsedArraysIn env kbody
353
354       -- do local reduction
```

```
355       let tid = fst $ head $ unSegSpace space
356       let env' = updateEnvTid env tid
357       reds <- lift $ do mkIntmRed env' kbody ts binops
358       let numResConsumed = numArgsConsumedBySegop binops
359       let (scanReds, fusedReds) = L.splitAt numResConsumed reds
360
361       -- scan over reduction results
362       imScan <- lift . buildSegScan "scan_agg" $ do
363         tid' <- newVName "tid"
364         let env'' = updateEnvTid env tid'
365         phys <- newVName "phys_tid"
366         binops' <- renameSegBinOp binops
367
368         let lvl' = SegThread SegNoVirt Nothing
369         let space' = SegSpace phys [(tid', grpSize env'')]
370         results <- mapM (buildKernelResult env'') scanReds
371         let ts' = L.take numResConsumed ts
372         pure (results, lvl', space', binops', ts')
373
374       scans' <- lift . buildSegMapTup_ "scan_res" $ do
375         tid' <- newVName "tid"
376         phys <- newVName "phys_tid"
377
378         let neutrals = L.map segBinOpNeutral binops
379         scanLambdas <- mapM (renameLambda . segBinOpLambda) binops
380
381         let scanNames = L.map getVName imScan
382
383         idx <- letSubExp "idx" =<< eBinOp (Sub Int64 OverflowUndef)
384                                           (eSubExp $ Var tid')
385                                           (eSubExp $ intConst Int64 1)
386         nes <- forM neutrals  (\n -> letTupExp' "ne" =<< eIf (eCmpOp (
    CmpEq $ IntType Int64)
387                                             (eSubExp $ Var tid')
388                                             (eSubExp $ intConst Int64 0)
389                                           )
390                                           (eBody $ L.map toExp n)
391                                             (eBody $ L.map (\s -> eIndex s [
    eSubExp idx]) scanNames))
392
393         let tidMap = M.singleton tid tid'
394         let kbody' = substituteNames tidMap kbody
```

```
395         iot <- buildSeqFactorIota env
396         let env'' = updateEnvTid env tid'
397         lambSOAC <- buildSOACLambda env'' usedArrays iot kbody' ts
398         let scans = L.map (\(l, n) -> Scan l n) $ L.zip scanLambdas nes
399         let scanSoac = scanomapSOAC scans lambSOAC
400         es <- mapM (getChunk env'') usedArrays
401         res <- letTupExp' "res" $ Op $ OtherOp $ Screma (seqFactor env)
      (es ++ [iot]) scanSoac
402         let usedRes = L.map (Returns ResultMaySimplify mempty) $ L.take
     numResConsumed res
403         fused <- mapM (buildKernelResult env'') fusedReds
404
405         let lvl' = SegThread SegNoVirt Nothing
406         let space' = SegSpace phys [(tid', grpSize env)]
407         let types' = scremaType (seqFactor env) scanSoac
408         pure (usedRes ++ fused, lvl', space', types')
409
410       tps <- mapM lookupType scans'
411       let shapes = L.map arrayShape tps
412       let shapes' = L.map (\s -> setOuterDims s 2 (Shape [grpsizeOld
     env])) shapes
413       let tps' = L.map (\(t, s) -> setArrayShape t s) (L.zip tps shapes
     ')
414       -- let tps' = mapM setArrayShape tps (Shape [grpsizeOld env])
415       pat' <- updateSegOpPatTypes 0 pat tps'
416       lift $
417         do forM_ (L.zip (patElems pat') scans') (\(p, s) -> do
418             let shape = arrayShape $ patElemType p
419             let exp' = BasicOp $ Reshape ReshapeArbitrary shape s
420             addStm $ Let (Pat [p]) aux exp')
421
422 seqStm' env (Let pat aux (Match scrutinee cases def dec)) = do
423   cases' <- forM cases seqCase
424   let (Body ddec dstms dres) = def
425   dstms' <- collectSeqBuilder' $ forM (stmsToList dstms) (seqStm' env)
426   (dres', stms') <- collectSeqBuilder $ localScope (scopeOf dstms') $
      fixReturnTypes pat dres
427   let def' = Body ddec (dstms' <> stms') dres'
428   lift $ do addStm $ Let pat aux (Match scrutinee cases' def' dec)
429   where
430     seqCase :: Case (Body GPU) -> SeqBuilder (Case (Body GPU))
431     seqCase (Case cpat body) = do
```

```
      let (Body bdec bstms bres) = body
      bstms' <- collectSeqBuilder' $
                  forM (stmsToList bstms) (seqStm' env)
      (bres', stms') <- collectSeqBuilder $ localScope (scopeOf bstms')
    $ fixReturnTypes pat bres
      let body' = Body bdec (bstms' <> stms') bres'
      pure $ Case cpat body'


seqStm' env (Let pat aux (Loop header form body)) = do
  let fparams = L.map fst header
  let (Body bdec bstms bres) = body
  bstms' <- collectSeqBuilder' $
              localScope (scopeOfFParams fparams) $
                forM_ (stmsToList bstms) (seqStm' env)
  (bres', stms') <- collectSeqBuilder $
                      localScope (scopeOf bstms') $
                        fixReturnTypes pat bres
  let body' = Body bdec (bstms' <> stms') bres'
  lift $ do addStm $ Let pat aux (Loop header form body')



-- Catch all
seqStm' _ stm = lift $ do addStm stm

-- Update types of pat to match the return types of an intermediate
   SegOp
-- first keep patterns will not be changed
updateSegOpPatTypes :: Int -> Pat Type -> [Type] -> SeqBuilder(Pat Type
   )
updateSegOpPatTypes keep pat tps = do
  let (patKeep, patUpdate) = L.splitAt keep $ patElems pat
  pure $ Pat $ patKeep ++
        L.map (\(p, t) -> setPatElemDec p t) (L.zip patUpdate (L.drop
   keep tps))



seqScatter :: Env -> Stm GPU -> SeqBuilder ()
seqScatter env (Let pat aux (Op (SegOp
              (SegMap (SegThread {}) space ts kbody))))
```

```
470   | L.length (unSegSpace space) /= 1 = throwError ()
471   | otherwise = do
472
473     -- Create the Loop expression
474     let (dests, upds) = L.unzip $ L.map (\(WriteReturns _ dest upds )
       -> (dest, upds)) (kernelBodyResult kbody)
475     loopInit <-
476         forM dests $ \d -> do
477             tp <- lookupType d
478             let decl = toDecl tp Unique
479             p <- newParam "loop_param" decl
480             pure (p, Var d)
481
482
483     -- Collect a set of all is and vs used in returns (only the Vars)
484     let upds' = L.concatMap (\ (slice, vs) -> do
485               let is = L.map (\ dim ->
486                                 case dim of
487                                   DimFix d -> d
488                                   _ -> error "please no"
489                     ) (unSlice slice)
490               vs : is
491             ) $ concat upds
492     let upd'' = L.filter (\u ->
493               case u of
494                 Var _ -> True
495                 _ -> False
496           ) upds'
497     let updNames = S.fromList $ L.map (\(Var n) -> n) upd''
498     -- Intersect it with all pattern names from the kbody
499     let names = S.fromList $ L.concatMap (\ (Let pat _ _) ->
500               patNames pat
501             ) $ kernelBodyStms kbody
502     -- The names that should have "producing statements"
503     let pStms = S.toList $ S.difference updNames names
504
505     let paramMap = M.fromList $ L.map invert loopInit
506
507     i <- newVName "loop_i"
508     let loopForm = ForLoop i Int64 (seqFactor env)
509
510     body <- lift . buildBody_ $ do
```

```
511
512         mapRes <- buildSegMapTup "map_res" $ do
513             let tid = fst $ head $ unSegSpace space
514             phys <- newVName "phys_tid"
515
516             -- size <- mkChunkSize tid env
517             offset <- letSubExp "offset" $ BasicOp $
518                         BinOp (Mul Int64 OverflowUndef) (Var tid) (
    seqFactor env)
519             tmp <- letSubExp "tmp" $ BasicOp $
520                         BinOp (Sub Int64 OverflowUndef) (grpsizeOld env
    ) offset
521             size <- letSubExp "size" $ BasicOp $
522                         BinOp (SMin Int64) tmp (seqFactor env)
523             size' <- letSubExp "size'" =<< eBinOp (Sub Int64
    OverflowUndef)
524                                                   (eSubExp size)
525                                                   (eSubExp $ intConst
    Int64 1)
526             i' <- letSubExp "loop_i'" $ BasicOp $
527                                       BinOp (SMin Int64) size' (Var i)
528             idx <- letSubExp "idx" =<< eBinOp (Add Int64 OverflowUndef)
529                                               (eSubExp i')
530                                               (eSubExp offset)
531
532             -- Modify original statements
533             forM_ (kernelBodyStms kbody) $ \ stm -> do
534                 case stm of
535                   (Let pat' aux' (BasicOp (Index arr _))) -> do
536                       let arr' = M.findWithDefault arr arr (nameMap env
    )
537                       tp' <- lookupType arr'
538                       let slice' = case arrayRank tp' of
539                                        1 -> Slice [DimFix idx]
540                                        2 -> Slice [DimFix $ Var tid,
    DimFix i']
541                                        _ -> error "Scatter more than two
     dimensions"
542                       addStm $ Let pat' aux' (BasicOp (Index arr' slice
    '))
543                   stm -> addStm stm
544
```

```
545             -- Potentially create more statements and create a mapping
    from the
546             -- original name to the new subExp
547             mapping <- forM pStms $ \ nm -> do
548                 offset <- letSubExp "iota_offset" =<< eBinOp (Mul Int64
     OverflowUndef)
549                                                              (eSubExp $
     Var tid)
550                                                              (eSubExp $
     seqFactor env)
551                 val <- letSubExp "iota_val" =<< eBinOp (Add Int64
    OverflowUndef)
552                                                        (eSubExp
    offset)
553                                                        (eSubExp i')
554                 pure (Var nm, val)
555             let valMap = M.fromList mapping
556
557
558             -- Update the original WriteReturns to target the loop
    params instead
559             res' <- forM (kernelBodyResult kbody) $ \ res -> do
560                 case res of
561                     (WriteReturns _ dest upd) -> do
562                         let (Just destParam) = M.lookup (Var dest)
    paramMap
563                         let dest' = paramName destParam
564                         let upd' = L.map (mapUpdates valMap) upd
565                         pure $ WriteReturns mempty dest' upd'
566                     _ -> error "Expected WriteReturns in scatter"
567
568             -- Return the results of the update statements form the
    segmap
569             let lvl' = SegThread SegNoVirt Nothing
570             let space' = SegSpace phys [(tid, grpSize env)]
571             -- let res' = L.map (Returns ResultMaySimplify mempty)
    updates
572             pure (res', lvl', space', ts)
573
574
575         -- Return the results from the segmap from the loop
576         let res = L.map (SubExpRes mempty) mapRes
```

```
577         pure res
578
579     -- Construct the final loop
580     let loopExp = Loop loopInit loopForm body
581
582     lift $ do addStm $ Let pat aux loopExp
583
584   where
585     invert (a,b) = (b,a)
586
587     mapUpdates :: M.Map SubExp SubExp -> (Slice SubExp, SubExp) -> (
      Slice SubExp, SubExp)
588     mapUpdates mapping (Slice dims, vs) = do
589       let vs' = M.findWithDefault vs vs mapping
590       let dims' = L.map (\d ->
591             case d of
592               DimFix d' -> DimFix $ M.findWithDefault d' d' mapping
593               d' -> d' -- should never happen
594             ) dims
595       (Slice dims', vs')
596
597
598 seqScatter _ stm = error $
599                    "SeqScatter error. Should be a map at thread level
     but got"
600                    ++ show stm
601
602 buildSeqFactorIota :: Env -> Builder GPU VName
603 buildSeqFactorIota env = do
604   letExp "seq_index_iota" $ BasicOp $
605     Iota (seqFactor env) (intConst Int64 0) (intConst Int64 1) Int64
606
607
608 -- | Fixes the type of results and returns new results
609 fixReturnTypes :: Pat (LetDec GPU) -> Result -> SeqBuilder Result
610 fixReturnTypes pat result = do
611   let pelems = patElems pat
612   let pairs = L.zip pelems result
613   mapM fix pairs
614   where
615     fix :: (PatElem Type, SubExpRes) -> SeqBuilder SubExpRes
616     fix (pelem, subres) = do
```

```
617       let pdec = patElemDec pelem
618       sdec <- subExpResType subres
619
620       -- If the types differ create some statements to fix it
621       if sdec == pdec then
622         pure subres
623       else
624         case subExpResVName subres of
625           Nothing -> pure subres -- constant just return original
626           Just name -> lift $ do
627             let newShape = arrayShape pdec
628             resSubExp <- letSubExp "res_falt" $ BasicOp $
629                          Reshape ReshapeArbitrary newShape name
630             pure $ subExpRes resSubExp
631
632 buildSOACLambda :: Env -> [VName] -> VName -> KernelBody GPU -> [Type]
      -> Builder GPU (Lambda GPU)
633 buildSOACLambda env usedArrs indexName kbody retTs = do
634   let tid = getThreadId env
635   ts <- mapM lookupType usedArrs
636   let ts' = L.map (Prim . elemType) ts
637   params <- mapM (newParam "par" ) ts'
638   itp <- lookupType indexName
639   indexParam <- newParam "par_index" $ Prim $ elemType itp
640   offset <- letSubExp "offset" =<< eBinOp (Mul Int64 OverflowUndef)
641                                           (eSubExp $ seqFactor env)
642                                           (eSubExp $ Var tid)
643   (gTid, stms') <- collectStms $ letExp "gtid" =<< eBinOp (Add Int64
       OverflowUndef)
644                                                           (eSubExp
       offset)
645                                                           (eSubExp $
       Var $ paramName indexParam)
646   let mapping = M.fromList $ L.zip (usedArrs ++ [tid]) $ L.map
       paramName params ++ [gTid]
647   let env' = updateMapping env mapping
648
649   kbody' <- runSeqMExtendedScope (seqKernelBody' env' kbody) (
       scopeOfLParams params)
650   let body = kbodyToBodyWithStms kbody' stms'
651   renameLambda $
652     Lambda
```

```
653     { lambdaParams = params ++ [indexParam],
654       lambdaBody = body,
655       lambdaReturnType = retTs
656     }
657
658 getVName :: SubExp -> VName
659 getVName (Var name) = name
660 getVName e = error $ "SubExp is not of type Var in getVName:\n" ++ show
        e
661
662 getTidIndexExp :: Env -> VName -> Builder GPU (Exp GPU)
663 getTidIndexExp env name = do
664   tp <- lookupType name
665   let outerDim = [DimFix $ Var $ getThreadId env]
666   let index =
667         case arrayRank tp of
668           0 -> SubExp $ Var name
669           1 -> Index name $ Slice outerDim
670           _ -> do
671             let dims = L.tail $ arrayDims tp
672             let innerDims = L.map (\d -> DimSlice (intConst Int64 0) d
        (intConst Int64 1)) dims
673             let allDims = outerDim ++ innerDims
674             Index name $ Slice allDims
675   pure $ BasicOp index
676
677 -- build a kernelresult from a single vName
678 buildKernelResult :: Env -> VName -> Builder GPU KernelResult
679 buildKernelResult env name = do
680   i <- getTidIndexExp env name
681   res <- letSubExp "res" i
682   pure $ Returns ResultMaySimplify mempty res
683
684 -- Creates a new kernelbody with the provided names as its results
685 mkResultKBody :: Env -> KernelBody GPU -> [VName] -> Builder GPU (
        KernelBody GPU)
686 mkResultKBody env (KernelBody dec _ _) names = do
687   (res, stms) <- collectStms $ do mapM (buildKernelResult env) names
688   pure $ KernelBody dec stms res
689
690 -- get the number of results consumed by a segop
691 -- i.e. the number of non-map results fed into the binops of the segop
```

```
692 numArgsConsumedBySegop :: [SegBinOp GPU] -> Int
693 numArgsConsumedBySegop binops =
694   let numResUsed = L.foldl
695                    (\acc (SegBinOp _ (Lambda pars _ _) neuts _)
696                      -> acc + L.length pars - L.length neuts) 0 binops
697   in numResUsed
698
699 seqKernelBody' ::
700   Env ->
701   KernelBody GPU ->
702   SeqM (KernelBody GPU)
703 seqKernelBody' env (KernelBody dec stms results) = do
704   stms' <- seqStms'' env stms
705   pure $ KernelBody dec stms' results
706
707 seqStms'' ::
708   Env ->
709   Stms GPU ->
710   SeqM (Stms GPU)
711 seqStms'' env stms = do
712   foldM (\ss s -> do
713       ss' <- runBuilder_ $ localScope (scopeOf ss <> scopeOf s) $
    seqStm'' env s
714       pure $ ss <> ss'
715       ) mempty (stmsToList stms)
716
717 seqStm'' ::
718   Env ->
719   Stm GPU ->
720   Builder GPU ()
721 seqStm'' env (Let pat aux (BasicOp (Index arr _)))
722   | memberMapping env arr = do
723   let (Just name) = lookupMapping env arr
724   i <- getTidIndexExp env name
725   addStm $ Let pat aux i
726
727 seqStm'' env stm = do
728   let tid = getThreadId env
729   case lookupMapping env tid of
730     Just gtid -> addStm $ substituteNames (M.singleton tid gtid) stm
731     Nothing -> addStm stm
732
```

```
733 -- create the intermediate reduction used in scan and reduce
734 mkIntmRed ::
735   Env ->
736   KernelBody GPU ->
737   [Type] ->                    -- segmap return types
738   [SegBinOp GPU] ->
739   Builder GPU [VName]
740 mkIntmRed env kbody retTs binops = do
741     let ne    = L.map segBinOpNeutral binops
742     lambda <- mapM (renameLambda . segBinOpLambda) binops
743
744     buildSegMapTup_ "red_intermediate" $ do
745       tid <- newVName "tid"
746       let env' = updateEnvTid env tid
747       phys <- newVName "phys_tid"
748       sz <- mkChunkSize tid env
749       usedArrs <- getUsedArraysIn env kbody
750       let tidMap = M.singleton (getThreadId env) tid
751       let kbody' = substituteNames tidMap kbody
752       iot <- buildSeqFactorIota env
753       lambSOAC <- buildSOACLambda env' usedArrs iot kbody' retTs
754       -- TODO analyze if any fused maps then produce reduce?
755       -- we build the reduce as a scan initially
756       let scans = L.map (\(l, n) -> Scan l n) $ L.zip lambda ne
757       let screma = scanomapSOAC scans lambSOAC
758       chunks <- mapM (getChunk env') usedArrs
759
760       res <- letTupExp' "res" $ Op $ OtherOp $
761                 Screma (seqFactor env) (chunks ++ [iot]) screma
762       let numRes = numArgsConsumedBySegop binops
763       let (scanRes, mapRes) = L.splitAt numRes res
764
765       -- get the reduction result from the scan
766       redIndex <- letSubExp "red_index" =<< eBinOp (Sub Int64
    OverflowUndef)
767                                                    (eSubExp sz)
768                                                    (eSubExp $ intConst
    Int64 1)
769       redRes <- forM scanRes (\r -> do
770           let rName = getVName r
771           rType <- lookupType rName
772           let rDims = L.tail $ arrayDims rType
```

```
773           let innerDims = L.map (\d -> DimSlice (intConst Int64 0) d (
      intConst Int64 1)) rDims
774           let newDims = DimFix redIndex : innerDims
775           letSubExp "red_res" $ BasicOp $ Index (getVName r) (Slice
      newDims)
776           )
777       let res' = redRes ++ mapRes
778       let lvl' = SegThread SegNoVirt Nothing
779       let space' = SegSpace phys [(tid, grpSize env)]
780       let kres = L.map (Returns ResultMaySimplify mempty) res'
781       types' <- mapM subExpType res'
782       pure (kres, lvl', space', types')
783
784 getUsedArraysIn ::
785   Env ->
786   KernelBody GPU ->
787   Builder GPU [VName]
788 getUsedArraysIn env kbody = do
789   scope <- askScope
790   let (arrays, _) = L.unzip $ M.toList $ M.filter isArray scope
791   let free = IM.elems $ namesIntMap $ freeIn kbody
792   let freeArrays = arrays `intersect` free
793   let arrays' =
794         L.map ( \ arr ->
795           if M.member arr (nameMap env) then
796             let (Just tile) = M.lookup arr (nameMap env)
797             in tile
798           else arr
799           ) freeArrays
800   pure arrays'
801
802
803 getChunk ::
804   Env ->
805   VName ->              -- Array to get chunk from
806   Builder GPU VName
807 getChunk env arr = do
808   let tid = getThreadId env
809   let sz = seqFactor env
810   tp <- lookupType arr
811   offset <- letSubExp "offset" =<< eBinOp (Mul Int64 OverflowUndef)
812                                           (eSubExp $ seqFactor env)
```

```
813                                          (eSubExp $ Var tid)
814   let dims =
815         case arrayRank tp of
816           1 -> [DimSlice offset sz (intConst Int64 1)]
817           2 -> [DimFix $ Var tid, DimSlice (intConst Int64 0) sz (
      intConst Int64 1)]
818           _ -> error "unhandled dims in getChunk"
819
820   letExp "chunk" $ BasicOp $ Index arr (Slice dims)
821
822 -- conver the kernelbody to a body prepended with the additional
      statements
823 kbodyToBodyWithStms :: KernelBody GPU -> Stms GPU -> Body GPU
824 kbodyToBodyWithStms (KernelBody dec stms res) stms' =
825   let res' = L.map (subExpRes . kernelResultSubExp) res
826   in Body
827     { bodyDec = dec,
828       bodyStms = stms' >< stms,
829       bodyResult = res'
830     }
831
832
833 flattenResults ::
834   Pat (LetDec GPU)->
835   [KernelResult] ->
836   Builder GPU [KernelResult]
837 flattenResults pat kresults = do
838   subExps <- forM (L.zip kresults $ patTypes pat) $ \(res, tp)-> do
839     let resSubExp = kernelResultSubExp res
840     case resSubExp of
841       (Constant _) -> letSubExp "const_res" $ BasicOp $ SubExp
      resSubExp
842       (Var name) -> do
843           resType <- lookupType name
844           if arrayRank resType == 0 then
845             letSubExp "scalar_res" $ BasicOp $ SubExp resSubExp
846           else
847             letSubExp "reshaped_res" $ BasicOp $
848                         Reshape ReshapeArbitrary (arrayShape $
      stripArray 1 tp) name
849
850   let kresults' = L.map (Returns ResultMaySimplify mempty) subExps
```

```
851
852   pure kresults'
853
854 renameSegBinOp :: [SegBinOp GPU] -> Builder GPU [SegBinOp GPU]
855 renameSegBinOp segbinops =
856   forM segbinops $ \(SegBinOp comm lam ne shape) -> do
857     lam' <- renameLambda lam
858     pure $ SegBinOp comm lam' ne shape
859
860 -- Generates statements that compute the pr. thread chunk size. This is
        needed
861 -- as the last thread in a block might not have seqFactor amount of
       elements
862 -- to read.
863 mkChunkSize ::
864   VName ->                -- The thread id
865   Env ->
866   Builder GPU SubExp      -- Returns the SubExp in which the size is
867 mkChunkSize tid env = do
868   offset <- letSubExp "offset" $ BasicOp $
869               BinOp (Mul Int64 OverflowUndef) (Var tid) (seqFactor env)
870   tmp <- letSubExp "tmp" $ BasicOp $
871               BinOp (Sub Int64 OverflowUndef) (grpsizeOld env) offset
872   letSubExp "size" $ BasicOp $
873               BinOp (SMin Int64) tmp (seqFactor env)
874
875
876 -- | Creates a tile for each array in scope at the time of caling it.
877 -- That is if called at the correct time it will create a tile for each
878 -- global array
879 mkTiles ::
880   Env ->
881   SeqBuilder Env
882 mkTiles env = do
883   scope <- askScope
884   let arrsInScope = M.toList $ M.filter isArray scope
885
886   tileSize <- lift $ do letSubExp "tile_size" =<< eBinOp (Mul Int64
        OverflowUndef)
887                                                   (eSubExp $ seqFactor env
       )
888                                                   (eSubExp $ grpSize env)
```

```
889
890   tiles <- forM arrsInScope $ \ (arrName, arrInfo) -> do
891     let tp = elemType $ typeOf arrInfo
892
893     tileScratch <- lift $ do letExp "tile_scratch" $ BasicOp $
894                                 Scratch tp [tileSize]
895
896     -- Check the type of the array to see if has been transposed
897     arrType <- lookupType arrName
898     let arrShape = arrayShape arrType
899     let dim = DimSlice (intConst Int64 0) (grpsizeOld env) (intConst
      Int64 1)
900     slice' <- case arrShape of
901                   (Shape [_]) -> pure $ Slice [dim]
902                   (Shape [n, _])
903                     | grpsizeOld env /= n -> pure $ Slice [DimFix $
      grpId env, dim]
904                     | otherwise -> pure $ Slice [dim, DimFix $ grpId
      env]
905                   _ -> throwError ()
906
907
908     tileSlice <- lift $ do letSubExp "tile_slice" $ BasicOp $ Index
      arrName slice'
909     tileStaging <- lift $ do letExp "tile_staging" $ BasicOp $
910                                 Update Unsafe tileScratch
911                                   (Slice [DimSlice (intConst Int64 0)
912                                                    (grpsizeOld env)
913                                                    (intConst Int64 1)]
914                                 ) tileSlice
915
916     -- Now read the chunks using a segmap
917     let (VName n _) = arrName
918     tile <- lift $ buildSegMap_ ("tile_" ++ nameToString n) $ do
919       tid <- newVName "tid"
920       phys <- newVName "phys"
921
922       start <- letSubExp "start" =<< eBinOp (Mul Int64 OverflowUndef)
923                                             (eSubExp $ Var tid)
924                                             (eSubExp $ seqFactor env)
925
```

```
926       let slice = Slice [DimSlice start (seqFactor env) (intConst Int64
      1)]
927       chunk <- letSubExp "chunk" $ BasicOp $ Index tileStaging slice
928
929       let lvl = SegThread SegNoVirt Nothing
930       let space = SegSpace phys [(tid, grpSize env)]
931       let types = [Array tp (Shape [seqFactor env]) NoUniqueness]
932       let res = [Returns ResultPrivate mempty chunk]
933       pure (res, lvl, space, types)
934
935
936     pure (arrName, tile)
937
938   pure $ setMapping env (M.fromList tiles)
939
940
941 isArray :: NameInfo GPU -> Bool
942 isArray info = arrayRank (typeOf info) > 0
943
944 -- Assumes the SegSpace to only have a single dimension
945 differentSize :: SegSpace -> Env -> Bool
946 differentSize space env =
947   let sz = snd $ head $ unSegSpace space
948   in sz /= grpsizeOld env
949
950 -- | Checks if a kernel body ends in only WriteReturns results as then
      it
951 -- must be the body of a scatter
952 isScatter :: KernelBody GPU -> Bool
953 isScatter (KernelBody _ _ res) =
954   L.all isWriteReturns res
955   where
956     isWriteReturns (WriteReturns {}) = True
957     isWriteReturns _ = False
958
959 buildSegMap' ::
960   SeqBuilder ([KernelResult], SegLevel, SegSpace, [Type]) ->
961   SeqBuilder (Exp GPU)
962 buildSegMap' (ExceptT m) = do
963   (tmp, stms) <- lift . collectStms $ m
964   case tmp of
965     Left _ -> throwError ()
```

```
966     Right (kres, lvl, space, ts) -> do
967       let kbody = KernelBody () stms kres
968       pure $ Op $ SegOp $ SegMap lvl space ts kbody
969
970 -- Builds a SegMap at thread level containing all bindings created in m
971 -- and returns the subExp which is the variable containing the result
972 buildSegMap ::
973   String ->
974   Builder GPU ([KernelResult], SegLevel, SegSpace, [Type]) ->
975   Builder GPU SubExp
976 buildSegMap name m = do
977   ((res, lvl, space, ts), stms) <- collectStms m
978   let kbody = KernelBody () stms res
979   letSubExp name $ Op $ SegOp $ SegMap lvl space ts kbody
980
981 -- Like buildSegMap but returns the VName instead of the actual
982 -- SubExp. Just for convenience
983 buildSegMap_ ::
984   String ->
985   Builder GPU ([KernelResult], SegLevel, SegSpace, [Type]) ->
986   Builder GPU VName
987 buildSegMap_ name m = do
988   subExps <- buildSegMap name m
989   pure $ varFromExp subExps
990   where
991     varFromExp :: SubExp -> VName
992     varFromExp (Var nm) = nm
993     varFromExp e = error $ "Expected SubExp of type Var, but got:\n" ++
      show e
994
995 -- like buildSegMap but builds a tup exp
996 buildSegMapTup ::
997   String ->
998   Builder GPU ([KernelResult], SegLevel, SegSpace, [Type]) ->
999   Builder GPU [SubExp]
1000 buildSegMapTup name m = do
1001   ((res, lvl, space, ts), stms) <- collectStms m
1002   let kbody = KernelBody () stms res
1003   letTupExp' name $ Op $ SegOp $ SegMap lvl space ts kbody
1004
1005 -- Like buildSegMapTup but returns the VName instead of the actual
1006 -- SubExp. Just for convenience
```

```
buildSegMapTup_ ::
  String ->
  Builder GPU ([KernelResult], SegLevel, SegSpace, [Type]) ->
  Builder GPU [VName]
buildSegMapTup_ name m = do
  subExps <- buildSegMapTup name m
  pure $ L.map varFromExp subExps
  where
    varFromExp :: SubExp -> VName
    varFromExp (Var nm) = nm
    varFromExp e = error $ "Expected SubExp of type Var, but got:\n" ++
     show e

-- | The [KernelResult] from the input monad is what is being passed to
     the
-- segmented binops
buildSegScan ::
  String ->          -- SubExp name
  Builder GPU ([KernelResult], SegLevel, SegSpace, [SegBinOp GPU], [
    Type]) ->
  Builder GPU [SubExp]
buildSegScan name m = do
  ((results, lvl, space, bops, ts), stms) <- collectStms m
  let kbody = KernelBody () stms results
  letTupExp' name $ Op $ SegOp $ SegScan lvl space bops ts kbody
```

C Unit tests

C.1 Original

```
let main [n] [m] (a : [n][m]u32) =
  #[incremental_flattening(only_intra)]
  map (\ a_row ->
     map (\a -> a + 2) a_row
  ) a

-- Map-map-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[100000][1024]u32} auto output
-- compiled random input {[200000][1024]u32} auto output
-- compiled random input {[300000][1024]u32} auto output
```

```
-- compiled random input {[400000][1024]u32} auto output
-- compiled random input {[500000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) = main a

-- ==
-- entry: testThreads
-- compiled random input { [131072][1024]u32 }
-- compiled random input { [262144][512]u32 }
-- compiled random input { [524288][256]u32 }
-- compiled random input { [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) = main a
```

```
let main [n] [m] (a : [n][m]u32) =
  #[incremental_flattening(only_intra)]
  map (\ a_row ->
     reduce (+) 0 a_row
  ) a

-- Map-reduce-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[100000][1024]u32} auto output
-- compiled random input {[200000][1024]u32} auto output
-- compiled random input {[300000][1024]u32} auto output
-- compiled random input {[400000][1024]u32} auto output
-- compiled random input {[500000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) = main a

-- ==
-- entry: testThreads
-- compiled random input { [131072][1024]u32 }
-- compiled random input { [262144][512]u32 }
-- compiled random input { [524288][256]u32 }
-- compiled random input { [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) = main a
```

```
let main [n] [m] (a : [n][m]u32) =
  #[incremental_flattening(only_intra)]
  map (\ a_row ->
     scan (+) 0 a_row
  ) a

```

```
-- Map-scan-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[100000][1024]u32} auto output
-- compiled random input {[200000][1024]u32} auto output
-- compiled random input {[300000][1024]u32} auto output
-- compiled random input {[400000][1024]u32} auto output
-- compiled random input {[500000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) = main a

-- ==
-- entry: testThreads
-- compiled random input { [131072][1024]u32 }
-- compiled random input { [262144][512]u32 }
-- compiled random input { [524288][256]u32 }
-- compiled random input { [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) = main a
```

```
let main [n] [m] (dss: [n][m]u32) (vss: [n][m]u32) =
  #[incremental_flattening(only_intra)]
  map2 (\ ds vs ->
    scatter (copy ds) (iota m) vs
    ) dss vss

-- Map-scatter-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[50000][1024]u32 [50000][1024]u32} auto
   output
-- compiled random input {[100000][1024]u32 [100000][1024]u32} auto
   output
-- compiled random input {[150000][1024]u32 [150000][1024]u32} auto
   output
-- compiled random input {[200000][1024]u32 [200000][1024]u32} auto
   output
-- compiled random input {[250000][1024]u32 [250000][1024]u32} auto
   output
entry testBlocks [n] [m] (a : [n][m]u32) (c : [n][m]u32) = main a c

-- ==
-- entry: testThreads
-- compiled random input { [131072][1024]u32  [131072][1024]u32 } auto
```

```
    output
-- compiled random input { [262144][512]u32  [262144][512]u32 } auto
    output
-- compiled random input { [524288][256]u32  [524288][256]u32 } auto
    output
-- compiled random input { [1048576][128]u32  [1048576][128]u32 } auto
    output
entry testThreads [n] [m] (a : [n][m]u32) (c : [n][m]u32) = main a c
```

```
let main [n] [m] (dss: [n][m]u32) (iss: [n][m]i64) (vss: [n][m]u32) =
  #[incremental_flattening(only_intra)]
  map3 (\ ds is vs ->
    scatter (copy ds) is vs
    ) dss iss vss

-- Map-scatter-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[50000][1024]u32 [50000][1024]i64
    [50000][1024]u32} auto output
-- compiled random input {[100000][1024]u32 [100000][1024]i64
    [100000][1024]u32} auto output
-- compiled random input {[150000][1024]u32 [150000][1024]i64
    [150000][1024]u32} auto output
-- compiled random input {[200000][1024]u32 [200000][1024]i64
    [200000][1024]u32} auto output
-- compiled random input {[250000][1024]u32 [250000][1024]i64
    [250000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) (b : [n][m]i64) (c : [n][m]u32
    ) = main a b c

-- ==
-- entry: testThreads
-- compiled random input { [131072][1024]u32  [131072][1024]i64
    [131072][1024]u32 }
-- compiled random input { [262144][512]u32  [262144][512]i64
    [262144][512]u32 }
-- compiled random input { [524288][256]u32  [524288][256]i64
    [524288][256]u32 }
-- compiled random input { [1048576][128]u32  [1048576][128]i64
    [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) (b : [n][m]i64) (c : [n][m]
```

```
    u32) = main a b c
```

C.2 Sequential

```
let main [n] [m] (a : [n][m]u32) =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map (\ a_row ->
     map (\a -> a + 2) a_row
  ) a

-- Map-map-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[100000][1024]u32} auto output
-- compiled random input {[200000][1024]u32} auto output
-- compiled random input {[300000][1024]u32} auto output
-- compiled random input {[400000][1024]u32} auto output
-- compiled random input {[500000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) = main a

-- ==
-- entry: testThreads
-- compiled random input { [32768][4096]u32 }
-- compiled random input { [65536][2048]u32 }
-- compiled random input { [131072][1024]u32 }
-- compiled random input { [262144][512]u32 }
-- compiled random input { [524288][256]u32 }
-- compiled random input { [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) = main a
```

```
let main [n] [m] (a : [n][m]u32) =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map (\ a_row ->
     reduce (+) 0 a_row
  ) a

-- Map-reduce-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[100000][1024]u32} auto output
```

```
-- compiled random input {[200000][1024]u32} auto output
-- compiled random input {[300000][1024]u32} auto output
-- compiled random input {[400000][1024]u32} auto output
-- compiled random input {[500000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) = main a

-- ==
-- entry: testThreads
-- compiled random input { [32768][4096]u32 }
-- compiled random input { [65536][2048]u32 }
-- compiled random input { [131072][1024]u32 }
-- compiled random input { [262144][512]u32 }
-- compiled random input { [524288][256]u32 }
-- compiled random input { [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) = main a
```

```
let main [n] [m] (a : [n][m]u32) =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map (\ a_row ->
     scan (+) 0 a_row
  ) a

-- Map-scan-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[100000][1024]u32} auto output
-- compiled random input {[200000][1024]u32} auto output
-- compiled random input {[300000][1024]u32} auto output
-- compiled random input {[400000][1024]u32} auto output
-- compiled random input {[500000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) = main a

-- ==
-- entry: testThreads
-- compiled random input { [32768][4096]u32 }
-- compiled random input { [65536][2048]u32 }
-- compiled random input { [131072][1024]u32 }
-- compiled random input { [262144][512]u32 }
-- compiled random input { [524288][256]u32 }
-- compiled random input { [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) = main a
```

```
let main [n] [m] (dss: [n][m]u32) (vss: [n][m]u32) =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map2 (\ ds vs ->
    scatter (copy ds) (iota m) vs
    ) dss vss

-- Map-scatter-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[50000][1024]u32 [50000][1024]u32} auto output
-- compiled random input {[100000][1024]u32 [100000][1024]u32} auto output
-- compiled random input {[150000][1024]u32 [150000][1024]u32} auto output
-- compiled random input {[200000][1024]u32 [200000][1024]u32} auto output
-- compiled random input {[250000][1024]u32 [250000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) (c : [n][m]u32) = main a c

-- ==
-- entry: testThreads
-- compiled random input { [65536][2048]u32 [65536][2048]u32 } auto output
-- compiled random input { [131072][1024]u32  [131072][1024]u32 } auto output
-- compiled random input { [262144][512]u32  [262144][512]u32 } auto output
-- compiled random input { [524288][256]u32  [524288][256]u32 } auto output
-- compiled random input { [1048576][128]u32  [1048576][128]u32 } auto output
entry testThreads [n] [m] (a : [n][m]u32) (c : [n][m]u32) = main a c
```

```
let main [n] [m] (dss: [n][m]u32) (iss: [n][m]i64) (vss: [n][m]u32) =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map3 (\ ds is vs ->
    scatter (copy ds) is vs
```

```
    ) dss iss vss

-- Map-scatter-simple performance
-- ==
-- entry: testBlocks
-- compiled random input {[50000][1024]u32 [50000][1024]i64
   [50000][1024]u32} auto output
-- compiled random input {[100000][1024]u32 [100000][1024]i64
   [100000][1024]u32} auto output
-- compiled random input {[150000][1024]u32 [150000][1024]i64
   [150000][1024]u32} auto output
-- compiled random input {[200000][1024]u32 [200000][1024]i64
   [200000][1024]u32} auto output
-- compiled random input {[250000][1024]u32 [250000][1024]i64
   [250000][1024]u32} auto output
entry testBlocks [n] [m] (a : [n][m]u32) (b : [n][m]i64) (c : [n][m]u32
   ) = main a b c

-- ==
-- entry: testThreads
-- compiled random input { [65536][2048]u32 [65536][2048]i64
   [65536][2048]u32 }
-- compiled random input { [131072][1024]u32  [131072][1024]i64
   [131072][1024]u32 }
-- compiled random input { [262144][512]u32  [262144][512]i64
   [262144][512]u32 }
-- compiled random input { [524288][256]u32  [524288][256]i64
   [524288][256]u32 }
-- compiled random input { [1048576][128]u32  [1048576][128]i64
   [1048576][128]u32 }
entry testThreads [n] [m] (a : [n][m]u32) (b : [n][m]i64) (c : [n][m]
   u32) = main a b c
```

D Big Number Addition

```
-------------------------------------------------------------------------------

--- Implementation took heavy inspiration from:
--- [1] Amar Topalovic, Walter Restelli-Nielsen, Kristian Olesen:
---     ``Multiple-precision Integer Arithmetic'', DPP'22 final project
   ,
```

```
---     https://futhark-lang.org/student-projects/dpp21-mpint.pdf
-------------------------------------------------------------------------------

let imap2 as bs f = map2 f as bs

let imap2Seq f as bs =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map2 f as bs

let imap2Org f as bs =
  #[incremental_flattening(only_intra)]
  map2 f as bs
-------------------------------------------------------------------------------

---- prefix sum (scan) operator to propagate the carry
-- let add_op (ov1 : bool, mx1: bool) (ov2 : bool, mx2: bool) : (bool, bool) =
--   ( (ov1 && mx2) || ov2,    mx1 && mx2 )
-------------------------------------------------------------------------------


---- prefix sum (scan) operator to propagate the curry:
---- format: last digit set      => overfolow
----         ante-last digit set => one unit away from overflowing
let badd_op (c1 : u8) (c2: u8) : u8 =
  (c1 & c2 & 2) | (( (c1 & (c2 >> 1)) | c2) & 1)

let badd [n] (as : [n]u32) (bs : [n]u32) : [n]u32 =
  let (pres, cs) =
    imap2 as bs
      (\ a b -> let s = a + b
                let b = u8.bool (s < a)
                let b = b | ((u8.bool (s == u32.highest)) << 1)
                in  (s, b)
      ) |> unzip
  let carries = scan badd_op 2u8 cs
  in
    imap2 (iota n) pres
        (\ i r -> r + u32.bool (i > 0 && ( (#[unsafe] carries[i-1]) & 1u8 == 1u8)) )

```

```
-- Big-Integer Addition: performance
-- ==
-- entry: mainSeq
-- compiled random input { [131072][1024]u32    [131072][1024]u32 }
-- compiled random input { [262144][512]u32     [262144][512]u32 }
-- compiled random input { [524288][256]u32     [524288][256]u32 }
-- compiled random input { [1048576][128]u32    [1048576][128]u32 }

-- Big-Integer Addition: performance
-- ==
-- entry: mainOrg
-- compiled random input { [131072][1024]u32    [131072][1024]u32 }
-- compiled random input { [262144][512]u32     [262144][512]u32 }
-- compiled random input { [524288][256]u32     [524288][256]u32 }
-- compiled random input { [1048576][128]u32    [1048576][128]u32 }

-- computes one batched multiplication: a*b
entry mainSeq [m][n] (ass: [m][n]u32) (bss: [m][n]u32) : [m][n]u32 =
  imap2Seq badd ass bss

entry mainOrg [m][n] (ass: [m][n]u32) (bss: [m][n]u32) : [m][n]u32 =
  imap2Org badd ass bss
```

E Radix Futhark

```
let mapSeq as f =
  #[incremental_flattening(only_intra)]
  #[seq_factor(4)]
  map f as

let mapOrg as f =
  #[incremental_flattening(only_intra)]
  map f as

let partition2 [n] (p: u32 -> bool) (arr: [n]u32)
                : (i64, *[n]u32) =
   let cs = map p arr
   let tfs = map (\ f->if f then 1i64
                           else 0i64) cs
   let isT = scan (+) 0 tfs
   let i = isT[n-1]
```

```
17
18    let ffs = map (\f->if f then 0
19                          else 1) cs
20    let isF = map (+i) <| scan (+) 0 ffs
21    let inds = map (\(c,iT,iF) ->
22                      if c then iT-1
23                           else iF-1
24                  ) (zip3 cs isT isF)
25    let res = replicate n 0
26    in (i, scatter res inds arr)
27
28
29  let stepSeq [num_blocks] [num_elems] (digit : u32) (arr : *[num_blocks
        ][num_elems]u32)
30            : *[num_blocks][num_elems]u32 =
31     let b = 4
32     -- intra block sorting
33     let arr_intra = (mapSeq arr
34         (\row ->
35           let arr = loop row for j < b do
36               let (_, arr) = partition2 (\ele ->
37                 if (ele >> (digit*4 + (u32.i64 j))) & 1 == 0 then true
        else false
38               ) row
39               in arr
40           in arr
41         ))
42     in arr_intra
43
44  let stepOrg [num_blocks] [num_elems] (digit : u32) (arr : *[num_blocks
        ][num_elems]u32)
45            : *[num_blocks][num_elems]u32 =
46     let b = 4
47     -- intra block sorting
48     let arr_intra = (mapOrg arr
49         (\row ->
50           let arr = loop row for j < b do
51               let (_, arr) = partition2 (\ele ->
52                 if (ele >> (digit*4 + (u32.i64 j))) & 1 == 0 then true
        else false
53               ) row
54               in arr
```

```
55            in arr
56          ))
57      in arr_intra
58
59  -- Intra-group radix sort: performance
60  -- ==
61  -- entry: mainSeq
62  -- compiled random input { [65536][2048]u32 } auto output
63  -- compiled random input { [131072][1024]u32 } auto output
64  -- compiled random input { [262144][512]u32  } auto output
65  -- compiled random input { [524288][256]u32  } auto output
66  -- compiled random input { [1048576][128]u32 } auto output
67
68  -- Intra-group radix sort: performance
69  -- ==
70  -- entry: mainOrg
71  -- compiled random input { [131072][1024]u32  }
72  -- compiled random input { [262144][512]u32   }
73  -- compiled random input { [524288][256]u32   }
74  -- compiled random input { [1048576][128]u32  }
75
76  entry mainSeq [num_blocks] [num_elems] (arr : *[num_blocks][num_elems]
        u32)
77            : *[num_blocks][num_elems]u32 =
78      let num_digits = 8
79      in loop arr for i < num_digits do
80        stepSeq (u32.i32 i) arr
81
82  entry mainOrg [num_blocks] [num_elems] (arr : *[num_blocks][num_elems]
        u32)
83            : *[num_blocks][num_elems]u32 =
84      let num_digits = 8
85      in loop arr for i < num_digits do
86        stepOrg (u32.i32 i) arr
```

F Initial Benchmark Results

F.1 Without Sequentialization Optimization

```
Reporting arithmetic mean runtime of at least 10 runs for each dataset (min 0.5s).
More runs automatically performed for up to 300s to ensure accurate measurement.

../CUDA/bench/chain1.fut:fun1 (no tuning file):
[100000][1024]i64:       3572μs (95% CI: [    3570.8,     3573.4])
```

```
[100000][1023]i64:       3574 s (95% CI: [   3573.6,     3575.7])

../CUDA/bench/chain2.fut:fun1 (no tuning file):
[100000][1024]i64:       3787 s (95% CI: [   3780.2,     3792.3])
[100000][1023]i64:       3892 s (95% CI: [   3882.5,     3902.0])

../CUDA/bench/map reduce fused.fut (no tuning file):
[100000][1024]i64 [1024]i64:       2370 s (95% CI: [   2368.3,     2373.2])
[100000][1023]i64 [1023]i64:       2414 s (95% CI: [   2411.1,     2415.8])

../CUDA/bench/map reduce pre simple.fut (no tuning file):
[100000][1024]i64 [1024]i64:       1477 s (95% CI: [   1476.0,     1477.9])
[100000][1023]i64 [1023]i64:       1486 s (95% CI: [   1482.4,     1499.5])

../CUDA/bench/map reduce pre.fut (no tuning file):
[100000][1024]i64 [1024]i64:       2116 s (95% CI: [   2115.6,     2115.8])
[100000][1023]i64 [1023]i64:       2123 s (95% CI: [   2122.7,     2123.3])

../CUDA/bench/map reduce simple.fut (no tuning file):
[100000][1024]i64:       1265 s (95% CI: [   1264.2,     1266.0])
[100000][1023]i64:       1279 s (95% CI: [   1278.1,     1279.9])

../CUDA/bench/map reduce2.fut (no tuning file):
[100000][1024]i64:       2168 s (95% CI: [   2165.2,     2170.2])
[100000][1023]i64:       2208 s (95% CI: [   2205.8,     2208.9])

../CUDA/bench/map scan fused.fut (no tuning file):
[100000][1024]i64 [1024]i64:       5277 s (95% CI: [   5276.1,     5277.7])
[100000][1023]i64 [1023]i64:       5386 s (95% CI: [   5378.5,     5392.5])

../CUDA/bench/map scan pre simple.fut (no tuning file):
[100000][1024]i64 [1024]i64:       2853 s (95% CI: [   2852.9,     2853.4])
[100000][1023]i64 [1023]i64:       2884 s (95% CI: [   2883.6,     2885.9])

../CUDA/bench/map scan pre.fut (no tuning file):
[100000][1024]i64 [1024]i64:       4934 s (95% CI: [   4928.8,     4938.3])
[100000][1023]i64 [1023]i64:       5035 s (95% CI: [   5028.8,     5042.1])

../CUDA/bench/map scan simple.fut (no tuning file):
[100000][1024]i64:       2433 s (95% CI: [   2431.4,     2433.7])
[100000][1023]i64:       2441 s (95% CI: [   2436.4,     2444.1])

../CUDA/bench/scatter simple.fut (no tuning file):
[100000][1024]i16 [100000][1024]i64 [100000][1024]i16:       1024 s (95% CI: [   1024.1,     1024.4])
[100000][1023]i16 [100000][1023]i64 [100000][1023]i16:       1048 s (95% CI: [   1048.1,     1049.6])
[100000][999]i16 [100000][999]i64 [100000][999]i16:       1037 s (95% CI: [   1036.7,     1036.9])

../CUDA/bench/uses1.fut (no tuning file):
[100000][1024]i64 [1024]i64:       4435 s (95% CI: [   4434.5,     4435.0])
[100000][1023]i64 [1023]i64:       4605 s (95% CI: [   4599.7,     4608.6])
```

F.2 With Sequentialization Optimization

```
Reporting arithmetic mean runtime of at least 10 runs for each dataset (min 0.5 s).
More runs automatically performed for up to 300 s to ensure accurate measurement.

../CUDA/bench/chain1.fut:fun1 (no tuning file):
[100000][1024]i64:       2034 s (95% CI: [   2027.6,     2049.5])
[100000][1023]i64:       2026 s (95% CI: [   2025.5,     2025.8])

../CUDA/bench/chain2.fut:fun1 (no tuning file):
[100000][1024]i64:       2377 s (95% CI: [   2377.2,     2377.6])
[100000][1023]i64:       2422 s (95% CI: [   2421.4,     2421.9])

../CUDA/bench/map reduce fused.fut (no tuning file):
[100000][1024]i64 [1024]i64:       2112 s (95% CI: [   2111.4,     2112.0])
[100000][1023]i64 [1023]i64:       2172 s (95% CI: [   2168.7,     2175.3])

../CUDA/bench/map reduce pre simple.fut (no tuning file):
[100000][1024]i64 [1024]i64:       1171 s (95% CI: [   1168.9,     1181.5])
[100000][1023]i64 [1023]i64:       1180 s (95% CI: [   1178.9,     1181.5])

../CUDA/bench/map reduce pre.fut (no tuning file):
[100000][1024]i64 [1024]i64:       1573 s (95% CI: [   1572.1,     1574.1])
[100000][1023]i64 [1023]i64:       1600 s (95% CI: [   1599.1,     1601.5])

../CUDA/bench/map reduce simple.fut (no tuning file):
[100000][1024]i64:        737 s (95% CI: [    735.2,      738.3])
[100000][1023]i64:        755 s (95% CI: [    752.9,      756.1])
```

```
../CUDA/bench/map-reduce2.fut (no tuning file):
[100000][1024]i64:       1627μs (95% CI: [   1624.9,     1637.3])
[100000][1023]i64:       1672μs (95% CI: [   1671.9,     1672.8])

../CUDA/bench/map-scan-fused.fut (no tuning file):
[100000][1024]i64 [1024]i64:       3259μs (95% CI: [   3258.9,     3259.3])
[100000][1023]i64 [1023]i64:       3359μs (95% CI: [   3356.0,     3367.5])

../CUDA/bench/map-scan-pre-simple.fut (no tuning file):
[100000][1024]i64 [1024]i64:       1855μs (95% CI: [   1852.8,     1856.3])
[100000][1023]i64 [1023]i64:       1911μs (95% CI: [   1907.4,     1914.6])

../CUDA/bench/map-scan-pre.fut (no tuning file):
[100000][1024]i64 [1024]i64:       2717μs (95% CI: [   2713.9,     2718.8])
[100000][1023]i64 [1023]i64:       2768μs (95% CI: [   2767.4,     2767.8])

../CUDA/bench/map-scan-simple.fut (no tuning file):
[100000][1024]i64:       1447μs (95% CI: [   1444.1,     1449.5])
[100000][1023]i64:       1478μs (95% CI: [   1476.5,     1479.3])

../CUDA/bench/scatter-simple.fut (no tuning file):
[100000][1024]i16 [100000][1024]i64 [100000][1024]i16:       1291μs (95% CI: [   1289.4,     1293.2])
[100000][1023]i16 [100000][1023]i64 [100000][1023]i16:       1329μs (95% CI: [   1327.1,     1330.6])
[100000][999]i16 [100000][999]i64 [100000][999]i16:       1311μs (95% CI: [   1308.6,     1312.4])

../CUDA/bench/uses1.fut (no tuning file):
[100000][1024]i64 [1024]i64:       3258μs (95% CI: [   3253.0,     3262.8])
[100000][1023]i64 [1023]i64:       3325μs (95% CI: [   3322.5,     3327.1])
```