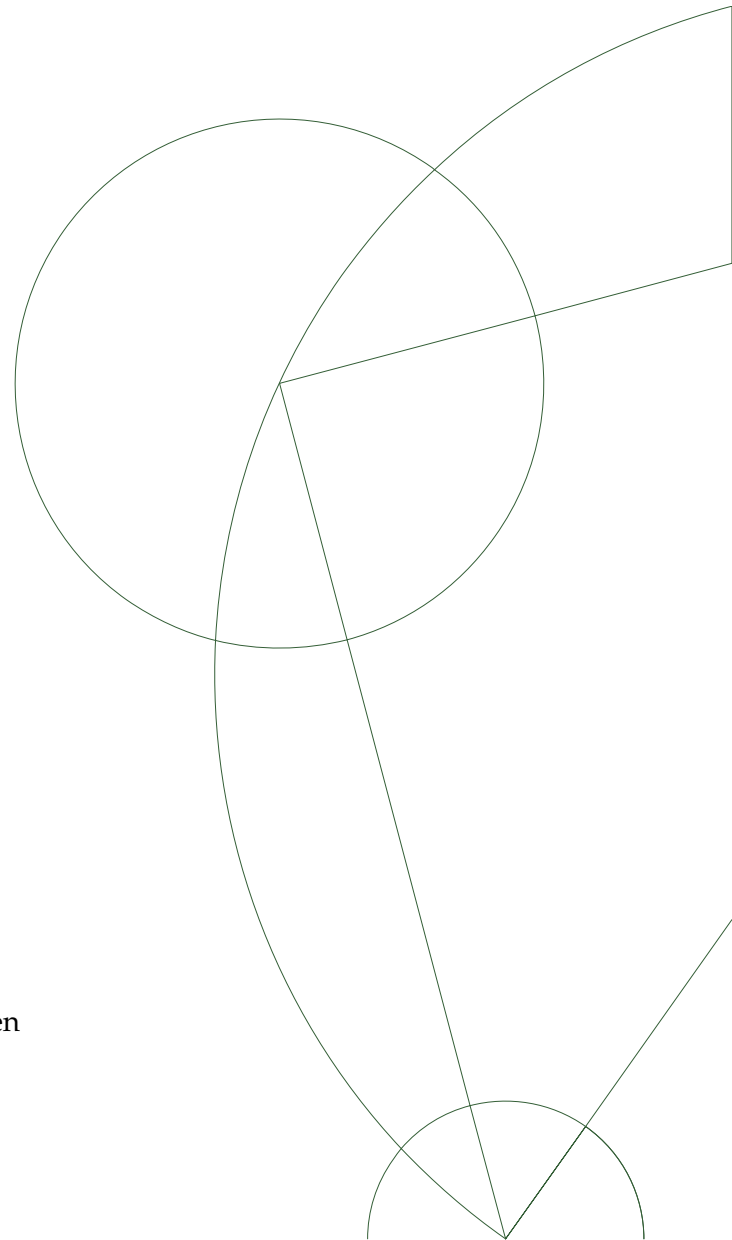
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Rasmus Wriedt Larsen – rasmuswriedtlarsen@gmail.com

Generating Efficient Code for Futhark's Segmented Redomap



Supervisors: Cosmin Eugen Oancea and Troels Henriksen

March 2017

Abstract

The current code generation for computing *segmented reductions* – reducing the innermost array of a multidimensional array – in the compiler for the purely functional array programming language Futhark, is suboptimal. The Futhark compiler will use a *segmented scan* to compute segmented reductions, which requires us to store all the intermediate steps of computing the reduction for each segment. As Futhark aims at achieving high performance on general purpose graphics processing units (GPGPUs), using this technique is a problem: the performance of most reductions are in nature constrained by the memory bandwidth of the GPU device, so writing all intermediate results back to memory will be inefficient.

Futhark has a special internal construct to represent the fusion of a reduction on the result of a map, called a *redomap*. For *segmented redomaps* – using a redomap within a map – the Futhark compiler will use a single GPU thread to sequentially compute the redomap for a whole segment. This can give very good performance when we launch so many threads that we fully utilize the GPU device; however, when there are only few segments with many elements, this technique is utterly useless.

In this thesis, we explore how to extend the Futhark compiler to generate efficient code for both segmented reductions and segmented redomaps. As Futhark requires that all multidimensional arrays must be *regular*, all segments must have the same size. We exploit this fact, and create multiple GPU kernels that are specialized to some configurations of number of segments and segment sizes. At runtime, we can decide which kernel will be most suitable, given a configuration of number of segments and segment size.

We study the performance of this implementation using simple performance experiments. For these experiments, the new implementation has significantly better performance than the two baseline approaches outlined above. We evaluate the implementation on several benchmarks ported from the Rodinia and Parboil benchmark suites; However, only the Backprop and K-means benchmarks from Rodinia shows any speedup, due to the fact that not all of the benchmarks have significant computations within a segmented reduction or a segmented redomap. From evaluation on four different GPUs, we can demonstrate a speedup, over the unmodified Futhark compiler, by a harmonic-mean factor of $1.39\times$ for Backprop, and by a harmonic-mean factor of $1.36\times$ for K-means.

Contents

Chapter 1

Introduction

Futhark is a purely functional array programming language, created with the aim of getting high performance on general purpose graphics processing units (GPGPUs) [20]. Futhark supports nested data parallelism by bulk array operators such as `map`, `reduce`, and `scan`. Futhark requires that all multidimensional arrays (also intermediate values) must be *regular*, meaning that all dimensions have a fixed size.

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

A common pattern is to perform a reduction on each inner array of a multidimensional array. This is called a *segmented reduction*, and an example computing the sum of each row of the 2D array `xss` can be seen above – we call this a segmented sum. We use the term *segment size* to refer to the size of the inner array being reduced (in our example `xs`), and *number of segments* to refer to the number of such arrays (and thereby also the number of results). As an example, if we want to compute a reduction over the innermost array of a 4D array with shape $[dim_0][dim_1][dim_2][dim_3]$, the number of segments will be $dim_0 \times dim_1 \times dim_2$ and the segment size will be dim_3.

```
reduce (+) 0 (map f xs)
```

Another common pattern is to perform a reduction on the result of a map, such as computing the sum of the result of applying the function `f` to all elements of `xs`, as can be seen in the example above. This pattern is recognized by the Futhark compiler and turned into a special construct, called a *redomap* [18], that will effectively compute both the reduction and the application of `f` in one pass over the data in `xs`. This is a significant improvement as both maps and reductions are typically memory bound computations (in subsection 3.3.4 we will study the redomap construct in more detail). Just as with reductions, we often see the pattern of a *segmented redomap*.

The current code generation for segmented reductions is suboptimal: the Futhark compiler will create a *segmented scan* (which computes an array containing all the intermediate results of the reduction), followed by writing the last element of each segment into an array, to compute the result of the segmented

reduction. Intuitively, as the segmented scan requires us to store all intermediate results, there must be a more efficient implementation strategy; furthermore, the current implementation of segmented scan in the Futhark compiler does not exploit the fact that arrays must be regular, a fact that should allow for optimizations in the generated code.

```
-- xss has shape [m][n]
map (\xs ->
        loop (sum = 0) = for i < n do
            sum + xs[i]
        in sum
    ) xss
```

The code generated by the Futhark compiler for a segmented redomap, will use a single GPU thread to sequentially process a whole segment. This strategy can be very efficient when we launch so many threads that we fully utilize the GPU, but will be very inefficient when there are only few segments with many elements. It is possible for a programmer to implement this strategy manually in Futhark for segmented reductions; the program code above does so for computing a segmented sum. As this strategy is expressed using a loop within a map, we will call this *loop-in-map*.

To sum up: when I started my thesis, there were two ways to compute segmented reductions and segmented redomaps; either using a segmented scan or a sequential loop. The main goal of my thesis is to extend the Futhark compiler with a more efficient method for computing segmented reductions and segmented redomaps, over all configuration of number of segments and segment size. The baseline we will compare my implementation against is therefore a segmented scan and a sequential loop. For the same number of total elements, I also hope to be able to compute a segmented reduction with the same efficiency as a normal one-dimensional reduction.

To examine the performance of the segmented scan and loop-in-map implementations, I have made a simple benchmark computing the segmented sum of a 2D-array. This is held against the performance of a reduction computing the sum of a 1D-array. We should take note of the fact that comparing a segmented reduction with a one-dimensional reduction is inherently skewed: when there are more segments, we need to write more results to memory, which increases the best possible runtime (we will examine this in more detail in subsection 5.2.1). I will also point out that the loop-in-map approach needs to transpose the input array to effectively access memory on the GPU, while the reduction of the one-dimensional array does not need this (we will cover this in section 4.4).

Figure 1.1 shows the results of reducing 2^{26} 32-bit floats, total of 256 MiB, on a NVIDIA GTX 780 Ti. The x-axis corresponds to different configurations of number of segments and segment size. We can see that the performance of the segmented-scan is far from the performance of the one-dimensional reduction, but otherwise pretty stable. The loop-in-map is really slow when there are too few segments to work on (off the charts), as a small number of threads have to do all the work. The loop-in-map strategy becomes really fast when there are enough segments that it can fully utilize the hardware (the GTX 780 Ti can have

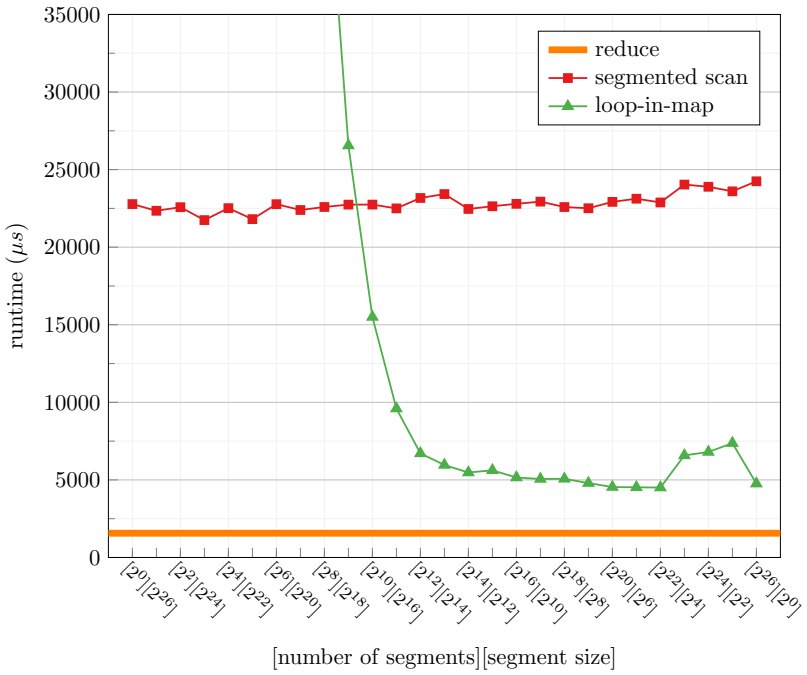


Figure 1.1: Performance of a computing segmented sum implemented as a segmented scan, a sequential loop within a map over the segments (loop-in-map). We use 2^{26} floats in different configurations of number of segments and segment size. We compare the performance with a 1D reduction computing the sum of as many elements (reduce). Run on a GTX 780 Ti.

30720 active threads[1]). As the segment size becomes very small, we can see the runtime takes a small hit in three cases: this is caused by the transposition, which has a bit worse performance in these cases.

The overall plan is to use three different strategies to effectively cover all cases of number of segments and segment size. (1) When there are few large segments we will use an approach similar to a one-dimensional reduction, where we use many groups of threads to cooperatively perform the reduction over a single segment. (2) When there are so many segments that we can fully utilize the GPU by using the loop-in-map approach, where we compute the reduction for a whole segment in a single thread, we will do so. Finally, (3) when there are not enough segments that using loop-in-map is effective, but the segment sizes are small, we will use a single group of threads to cooperatively reduce multiple segments. In this thesis, we will explore how to implement these strategies, and study their performance:

- I give the semantics of Futhark's reduction and redomap, and give an intuition of why they have an efficient parallel implementation (chapter 3). I will also give an introduction to the Futhark language, and present how a

[1] 2048 active threads on each of its 15 Streaming Multiprocessors

segmented scan can be implemented in the Futhark source language; this allows us to use the segmented scan to implement a segmented reduction in the Futhark source language.

- I summarize the most important factors for achieving good performance when programming for GPUs, with a detailed example for how reductions can be effectively implemented in CUDA (chapter 4). I will show how a transpose kernel can be implemented in CUDA, and why a transpose is needed for the loop-in-map strategy to have the optimal memory access pattern for a GPU.

- I present a prototype implementation for the three strategies outlined above, and experiments showing the raw performance characteristics of all three strategies (chapter 5).

  I will show the cost of performing a transpose, and briefly introduce a new transpose algorithm that will improve performance of the standard transpose algorithm when either the width or the height of the input array is low (subsection 5.2.4).

  I will show the performance of computing a commutative and a non-commutative segmented reduction using all three implementation strategies. I present a simple algorithm to decide which strategy to use for a given configuration of number of segments and segment size, and discuss how this algorithm could be extended with additional parameters to allow an autotuning project to tune which strategy is used (section 5.3).

- I give an overview of how my implementation in the Futhark compiler will generate code for segmented reductions and segmented redomaps (chapter 6). I will also discuss which cases can and cannot be handled by my implementation.

  I present a small performance evaluation of my implementation using segmented sum, and a non-trivial segmented reduction, showing that my implementation achieves performance better or equal to the two baseline approaches (section 6.5).

- I evaluate the performance improvements from using my implementation for segmented reductions and segmented redomaps, on ported benchmarks from the Rodinia and Parboil benchmark suites (chapter 7). The speedup is heavily influenced by the percentage of total work that is performed by segmented reductions and segmented redomaps; for the Backprop and K-means benchmarks from Rodinia, using 4 different GPUs, I can demonstrate a speedup by a harmonic-mean factor of $1.39\times$ and $1.36\times$ respectively, compared to the unmodified Futhark compiler.

- I present a survey on existing work for executing segmented reductions effectively on GPUs, and compare them with my implementation (chapter 8). I show a simple performance evaluation using segmented sum, that demonstrates that my implementation is able to get better performance than the solutions from Thrust, and Modern GPU.

Chapter 2

Notation and Naming Convention

Here are some small remarks on notation and naming convention I use in this thesis.

- For declaring the size of input data I use kibibytes where 1024 bytes = $1\,\mathrm{KiB}$, mebibytes where 1024^2 bytes = $1024\,\mathrm{KiB} = 1\,\mathrm{MiB}$, and gibibytes where 1024^3 bytes = $1024\,\mathrm{MiB} = 1\,\mathrm{GiB}$.
- I will often talk about time in microseconds, which is denoted by the symbol µs. The conversion ratio to milliseconds is $1000\,\mu\mathrm{s} = 1\,\mathrm{ms}$, and the conversion ratio to seconds is $1\,000\,000\,\mu\mathrm{s} = 1000\,\mathrm{ms} = 1\,\mathrm{s}$.
- One of the things that make GPU programming confusing, is that there are different terminology depending on whether you use CUDA or OpenCL. When introducing a GPU concept in chapter 4, I have striven to define both CUDA and OpenCL terms. However, I will always use the term *thread* instead of OpenCL's *work-item*, and most of the time use the term *group* instead of CUDA's *block* (but not when explaining CUDA code).
- As a reduction can be considered a redomap with the identity function as the "map part", I will often only say redomap if I talk about both a reduction and a redomap, and say reduction to mean only reduction (and not redomap).
- I will mark long word from a programming language and small segments of inline code as `some_inline_code(foo, bar)`.

Chapter 3

The Futhark Language and its redomap Construct

In this chapter I will give an introduction to the Futhark language, going in detail with the reduce and redomap constructs. I will only cover the parts needed to understand my thesis. For a more details of the Futhark language, please see [11, 12, 17, 18, 19, 20].

3.1 Notation

I will adopt the notation from other Futhark papers [18]. When q is a name for an object of some type, I will denote n of these as $\bar{q}^{(n)} = q_1, \ldots, q_n$, and $\bar{q} = q_1, \ldots, q_n$ for some n. I will also define $a[i]$ to be the element at index i of the array a (starting at index 0), and $\bar{a}[i] = (a_1[i], \ldots, a_n[i])$.

3.2 The Futhark Language

Futhark is a purely functional array programming language, created with the aim of getting high performance on GPUs [20]. It supports nested data parallelism on regular arrays by parallel *second order array combinators* (SOACs) such as `map`, `reduce`, and `scan`. Futhark is eagerly evaluated, with call-by-value semantics. Futhark has been designed to be a fairly simple language, to more easily allow for powerful compiler optimizations.

Futhark programs are written in a *source language*; one of the first things the compiler does after parsing a program is to convert it into a representation using the internal *core language*. Figure 3.1 shows the abstract syntax for the subset of the internal core language important for this thesis. There are a few differences between the two languages, which I will mention below, but in general they are very similar.

$\bar{q}^{(n)}$	::=	$q_1, \ldots, q_n$	(notation)
$\bar{q}$	::=	$q_1, \ldots, q_n$ for some n	(notation)
x	::=	`id`	(variable name)
z	::=	`id` | `Const`	(variable name or scalar value)
k	::=	`Const` | $[k_1, \ldots, k_n]$	(scalar or array value)
f	::=	`id`	(function name)
p	::=	x `:` τ	(typed variable)
t	::=	`i8` | `i16` | `i32` | `i64` | `bool` | `f32` | `f64`	(basic type)
τ	::=	t | $[z_1, \ldots, z_n]t$	(size-dep n-dim array type)
$l, \oplus$	::=	`(\(`$\bar{p}$`):(`$\bar{\tau}$`) ->` e`)`	(anonymous function)
e	::=	x	(variable)
	|	k	(scalar or array value)
	|	$(\bar{z}^{(n)})$	(n-tuple exp)
	|	x`[`z_1`, ..., `z_n`]`	(array indexing)
	|	$\odot\, z$	(unop)
	|	$z_1 \otimes z_2$	(binop)
	|	f `(`$\bar{z}$`)`	(function-call)
	|	`if` z `then` e_1 `else` e_2	(if)
	|	`let (`$\bar{p}$`) =` e_1 `in` e_2	(let-binding)
	|	`iota` z	(the array $[0,\ldots,z-1]$)
	|	`map` l $(\bar{x}^{(n)})$	(n-ary map)
	|	`reduce` $\oplus$ $(\bar{z}^{(n)})$ $(\bar{x}^{(n)})$	(reduce with n-ary op)
	|	`reduceComm` $\oplus$ $(\bar{z}^{(n)})$ $(\bar{x}^{(n)})$	(commutative reduce with n-ary op)
	|	`scan` $\oplus$ $(\bar{z}^{(n)})$ $(\bar{x}^{(n)})$	(scan with n-ary op)
	|	`redomap` $\oplus$ l $(\bar{z}^{(m)})$ $(\bar{x}^{(n)})$	(fusion of a reduce on the result of a map)
	|	x`[`z_1`,...,`z_n`] =` z	(in-place update)
	|	`loop ((`$\bar{p}^{(n)}$`) =` $(\bar{z}^{(n)})$`) =` `for` $z' < z''$ `do` e	(sequential do-loop)
P	::=	`fun` f`(`$\bar{p}^{(n)}$`) : (`$\bar{\tau}$`) =` e	(named function def)
	|	$P\ P$ | ϵ	

Figure 3.1: Syntax of Futhark's core language.
Adapted from [18]

To summarize the information presented in Figure 3.1, a Futhark program consists of multiple functions, which takes arguments and evaluates and expression (or multiple expressions by using let bindings). Most of the Futhark expressions are similar to other functional programming languages, but I will present some details for some interesting cases below. We will also explore some properties that are important to Futhark, such as regularity of arrays and the array-of-tuples to tuple-of-arrays transformation.

For convenience instead of writing `let` x `=` e `in let` y `= ...` we can write `let` x `=` e `let` y `= ...`.

3.2.1 Regular Arrays

Futhark requires that all multidimensional arrays (also intermediate values) must be *regular*, this means that all inner arrays must have the same shape – that

array dimensions must have a fixed size. The array `[ [1,2], [3,4], [5,6] ]` is valid and has the type `[3][2]i32`, but `[ [1], [3,4] ]` is invalid because it is not regular.

3.2.2 Array-of-Tuples to Tuple-of-Arrays

The source language allows tuples and arrays of tuples, e.g. `[ (1,true),(1,false) ]` has type `[](i32,bool)`. Arrays of the same outer size can be combined into an array-of-tuples with the `zip` operator, and this can be undone by `unzip`.

An important transformation performed early on in the compiler pipeline is transforming array-of-tuples to tuple-of-arrays. This means that expressions and functions that take a tuple as argument in the source language will take multiple arguments in the core language – one for each of the values in the tuple. There are edge cases where some values expressible in the source language are no longer valid after applying this transformation; this is not important for this thesis, but more details can be found at [11].

The reason for wanting tuple-of-arrays instead of array-of-tuples is that this helps enable good memory access patterns on GPUs (memory coalescing, which is introduced in subsection 4.2.1), and is a generally applied technique in the GPU-programming domain.

In many other languages this transformation is called Array-of-Structures (AoS) to Structure-of-Arrays (SoA).

3.2.3 Loop

`loop` $(\bar{p}^{(n)}$ `=` $\bar{z}^{(n)})$ `= for` $z' < z''$ `do` e

The loop construct is a bit special for a functional language, but can be thought of as a tail-recursive function. The loop construct works as follows: (1) Bind the accumulator variables $\bar{p}^{(n)}$ to the initial values $\bar{z}^{(n)}$, and bind z' to 0; (2) if the condition ($z' < z''$) is false, the loop will end and the result will be the current values of $\bar{p}^{(n)}$; otherwise, (3) evaluate e (with z' and $\bar{p}^{(n)}$ in scope), bind $\bar{p}^{(n)}$ to the result from evaluating e, increase z' by 1, and goto step (2).

```
fun main(xs : [n]i32) : i32 =
  loop (sum = 0) = for i < n do
      sum + xs[i]
  in sum
```

An example of a loop can be found in the program above, which uses the loop in a non-idiomatic way to compute the sum of the input list. Note that the variable n will be bound to the length of the input list when entering the function.

3.2.4 Array Slicing

The grammar in Figure 3.1 does not show the full capabilities of indexing an array. In fact, we can use a slice in every position where an array index is valid.

In the source language a slice of `i:j:s` will take elements from i to j (excluded) with a stride of s. In the core language `i:n*s` will start at i and take n elements with a stride of s.

```
fun main(xss : [m][n]i32) : [][]i32 =
  xss[0:m:2, 0:n:3]
```

As an example, the program above (in the source language) will produce a 2D array, containing every third element from every second row.

3.2.5 In-place Update & Uniqueness

In-place updates is a special feature of Futhark, and can only happen when the original array is marked as *unique* at that point in the program. An array is unique if it, and all array variables it is aliased with, is not used on any following execution path. The type of a unique array is prefixed with an asterisk, such as `*[]i32`.

Simply put, array variables are aliased if they would share the same underlying memory. Aliasing can occur from simple expressions such as `let` x `: []i32 =` y or when accessing a slice of an array `let` x `: []i32 = y[0]`. By using the `copy` x construct, we can create a unique copy of a non-unique array.

The uniqueness types in Futhark is inspired by Clean [3], but is only important for this thesis because they allow us to perform in-place updates.

3.3 Second Order Array Combinators

Here I will give more details for some of the important SOACs that are used in the rest of my thesis, which are not self explanatory (such as map).

3.3.1 Terminology

We define a *reduction operator* to be an operator $\oplus : A \to A \to A$ which is (1) *associative*, so that for all elements x, y, and z in the domain A it must satisfy:

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

and (2) must have a neutral element e, which for all elements x in the domain A must satisfy:

$$e \oplus x = x \oplus e = x$$

A reduction operator can also be *commutative*, if for all elements x and y in the domain A it satisfies:

$$x \oplus y = y \oplus x$$

Floating Points

In the following we will use floating points as the element type of reductions; this is standard practice within the field of parallel computing, even though floating point arithmetic is not associative [13].

3.3.2 Reduce

For this discussion we will define `reduce` using a bit different symbols than in the grammar:

$$\texttt{reduce} \ \oplus \ (e_1, \ldots, e_n) \ \ (a_1, \ldots, a_n)$$

where $\bar{e} = (e_1, \ldots, e_n)$ is the neutral element, and $\bar{a} = (a_1, \ldots, a_n)$ is the arrays for the reduction.

The `reduce` expression is well typed if (1) all arrays in $\bar{a}$ have the same size w in the outermost dimension; and (2) the types of $\bar{e}$, the types of the elements of $\bar{a}$, and the types of the operands of $\oplus$ all match.

If $\oplus$ is an associative reduction operator with the neutral element $\bar{e}$, the `reduce` expression is well-defined, and the result of evaluating it will be:

$$\texttt{reduce} \ \oplus \ \bar{e} \ \ \bar{a} \ \ = \ \ \bar{e} \oplus \bar{a}[0] \oplus \bar{a}[1] \oplus \cdots \oplus \bar{a}[w-1]$$

so the result of reducing an empty list is the neutral element $\bar{e}$.

Parallel Execution

The requirement in Futhark that the reduction operator must be associative is different from the `fold` functions known from ML and Haskell, but this allows us to easily parallelize the computation; we can split the arrays $\bar{a}$ into as many parts as we want, run a reduction on each part, and finally reduce all the partial results. As an example we could split $\bar{a}$ into two parts, divided at the index k, reduce each part on a different processor, and finally reduce the partial results from the two processors:

$$\left(\bar{e} \oplus \bar{a}[0] \oplus \bar{a}[1] \oplus \cdots \oplus \bar{a}[k-1]\right) \ \oplus \ \left(\bar{e} \oplus \bar{a}[k] \oplus \bar{a}[k+1] \oplus \cdots \oplus \bar{a}[w-1]\right)$$

A commutative reduction can be marked by using the `reduceComm` keyword instead of `reduce`. When dealing with a commutative reduction, there is no restriction on the order in which we can reduce elements. As an example we could split $\bar{a}$ into two parts, divided by even and odd indexes, reduce each part on a different processor, and finally reduce the partial results from the two processors (assuming w is an even number):

$$\left(\bar{e} \oplus \bar{a}[0] \oplus \bar{a}[2] \oplus \cdots \oplus \bar{a}[w-2]\right) \ \oplus \ \left(\bar{e} \oplus \bar{a}[1] \oplus \bar{a}[3] \oplus \cdots \oplus \bar{a}[w-1]\right)$$

```
fun max(x: i32) (y: i32): i32 =
  if x > y then x else y

-- (best, left, right, total)
fun redOp((bx, lx, rx, tx): (i32,i32,i32,i32))
         ((by, ly, ry, ty): (i32,i32,i32,i32)): (i32,i32,i32,i32) =
  ( max bx (max by (rx + ly))
  , max lx (tx+ly)
  , max ry (rx+ty)
  , tsx + tsy)

fun mapOp (x: i32): (i32,i32,i32,i32) =
  (max x 0, max x 0, max x 0, x)

fun main(xs: []i32): i32 =
  let (x, _, _, _) = reduce redOp (0,0,0,0) (map mapOp xs)
  in x
```

Figure 3.2: Parallel Futhark Implementation of Maximum Segment Sum (MSS)

Maximum Segment Sum

An interesting example of a problem that can be solved using a non-commutative reduction is the maximum segment sum (MSS) problem. In the MSS problem we are given an array and need to find the largest sum of all possible contiguous subarrays. (Disclaimer: this example is used in many other places in the Futhark documentation)

The code in Figure 3.2 shows the Futhark implementation for MSS. After having reduced the elements of a contiguous subarray, the resulting tuple will contain the four values (b, l, r, t), where b is the best MSS seen so far, l is the best MSS starting from the "left" of the subarray, r is the best MSS starting from the "right" of the subarray, and t is the total sum of the whole subarray.

Many programming languages targeting GPUs does not allow you to define a non-commutative reduction operator, so in these cases Futhark is more expressive. We look more at this in chapter 8.

3.3.3 Scan

$$\texttt{scan}\ \oplus\ (e_1, \ldots, e_n)\ \ (a_1, \ldots, a_n)$$

The scan expression is very similar to the reduce expression, and can be thought of as computing an array containing all the intermediate results of performing a reduction over the same arguments. The scan operator $\oplus$ must also be associative, and $\bar{e}$ must be the corresponding neutral element. As before, all input arrays must have the same size w. The result of evaluating a scan expression will be the array:

$$\left[\quad (\bar{a}[0]) \quad , \quad (\bar{a}[0] \oplus \bar{a}[1]) \quad , \quad \ldots \quad , \quad (\bar{a}[0] \oplus \bar{a}[1] \oplus \cdots \oplus \bar{a}[w-1]) \quad \right]$$

The resulting array has the same size w as the input arrays. This type of scan is called an *inclusive scan*, as the input element at index i is used to compute the result at index i.

A scan can be efficiently implemented in parallel on a GPU [16]. The cost of executing a scan is intuitively larger than the cost of a reduction: For a scan we must perform more computation and also write many more results to memory (a significant cost on the GPU).

3.3.4 Redomap

$$\texttt{redomap}\ \oplus\ f\ \ (e_1, \ldots, e_m)\ \ (a_1, \ldots, a_n)$$

A redomap is intuitively a fusion of a reduction on the result of a `map`. It is not possible to use a redomap directly in the source language. To define the semantics of redomap, I will use the following equation:

$$\begin{aligned}\texttt{redomap}\ \oplus\ f\ \bar{e}^{(m)}\ \bar{a}^{(n)} \equiv\ & \texttt{let}\ \bar{b}^{(l)} = \texttt{map}\ g\ \bar{a}^{(n)} \\ & \texttt{let}\ \bar{c}^{(m)} = (b_1, \ldots, b_m) \\ & \texttt{let}\ \bar{d}^{(k)} = (b_{k-l}, \ldots, b_l) \\ & \texttt{in}\ \left(\texttt{reduce}\ \oplus\ \bar{e}^{(m)}\ \bar{c}^{(m)}\ \texttt{,}\ \bar{d}^{(k)}\right)\end{aligned}$$

The function f is called a *fold-function*, and will take as parameters (1) an accumulator value $\bar{x}^{(m)}$ for the reduction and (2) an input from $\bar{a}$ which we will call $\bar{y}^{(n)}$. We apply g to $\bar{y}^{(n)}$ and as a result get $\bar{z}^{(l)}$. Of these l values we will use the first m values to compute the new reduction accumulator, and also return the last k values of $\bar{z}^{(n)}$ as the result of the "map-part" (possibly none, possibly all).

$$\begin{aligned}f \equiv \backslash \bar{x}^{(m)}\ \ \bar{y}^{(n)}\ \texttt{->}\ & \texttt{let}\ \bar{z}^{(l)} = g\ \bar{y}^{(n)} \\ & \texttt{in}\ \left(\ \bar{x}^{(m)} \oplus (z_1, \ldots, z_m)\ ,\ (z_{l-k}, \ldots, z_l)\ \right)\end{aligned}$$

We will assume that the g function will return the value(s) used in the reduction as the first m return values, if this is not the case we can modify g to do so (e.g., if the reduction is over the unmodified elements of $\bar{a}^{(n)}$, we can make a function g' that adds $\bar{y}^{(n)}$ as the first return values).

A redomap can be either commutative or non-commutative, depending on the reduction it represents. We will denote a commutative redomap as `redomapComm`. Redomap has a very efficient parallel implementation, comparable to hand optimized Thrust code [18].

```
-- input program (source language)
fun main (ys : []i32) : (i32, []i32) =
  let ys_doubled = map (\y -> y*2) ys
  in (reduceComm (+) 0 ys, ys_doubled)

-- as redomap (core language)
fun main (ys : []i32) : (i32, []i32) =
  redomapComm (+)
              (\x y -> let z = y*2 in (x+y, z))
              0
              ys
```

An example of how the transformation to a redomap works can be seen above, with a program that computes both the sum of the input array `ys` but also an array with the result of multiplying all elements in `ys` with 2.

If we look back at the MSS example, this will be converted to a redomap by the compiler; This means that the result of `map mapOp xs` will never be manifested as an array in memory.

Naming Convention for Redomaps and Reductions

As a reduction can be considered as a redomap with g as the identity function, I will often only say redomap if I talk about both a reduction and a redomap, and say reduction to mean only reduction (and not redomap).

3.4 Segmented SOAC

We denote a SOAC as *segmented* if it is mapped over a multidimensional array, for example *segmented reduction* means `map (\xs -> reduce` $\oplus$ $\bar{e}$ `xs)`. In the same way we can have a *segmented scan* and a *segmented redomap*. Such segmented operations are not uncommon, for example they appear often in the translation of APL programs to Futhark [10, 21].

We can make a segmented version of the MSS program, by simply changing the `main` function to the following[1]:

```
fun main(xss: [m][n]i32): [m]i32 =
  map (\xs -> let (x, _, _, _) = reduce redOp (0,0,0,0) (map mapOp
      xs) in x) xss
```

As mentioned in the introduction, when the Futhark compiler encounters a segmented reduction it will generate code using first a segmented scan, followed by picking off the last element from each segment. For a segmented redomap, it will use one thread to process a segment sequentially. The main goal of this thesis is to improve on this state of affairs.

3.4.1 Implementing Segmented Scan

We can implement a segmented scan in a source language Futhark program by using a normal scan and a flag array, a method that was pioneered by NESL [5]. The code below uses this method to compute a segmented prefix-sum. I mention this example because it is the way a segmented scan is implemented in the Futhark compiler.

To mark the beginning of a segment, we use `true` as the flag value; all other flags should be set to `false` initially. In the flag-reduction operator, if the flag for the right operand is `true` this means that we are computing the result for an element in another segment, so we return the element from the right operand

[1] I named the dimension of the input array in this case, but we do not need to

unchanged. When combining values of the same segment, we will propagate the flag value of the left operand. This ensures that once an element has been combined with the first element of the segment (have gotten its final value) it will not be changed later.

```
fun seg_scan_sum_op (x_flag: bool, x: i32)
                    (y_flag :bool, y :i32) : (bool,i32) =
  if y_flag
  then (x_flag || y_flag, y)
  else (x_flag || y_flag, x + y)

fun segmented_scan_sum (flags: [n]bool, as: [n]i32): [n]i32 =
  let (_, res) = unzip (scan seg_scan_sum_op (false, 0) (zip flags
      as))
  in res
```

3.4.2 Implementing Segmented Reduction

We can now use this segmented scan to implement a segmented reduction. For a regular 2D array with shape $[m][n]$, we create the flag array by setting every n^{th} entry to `true` and the rest to `false`, we must make the input array flat, compute the segmented scan, and finally we must read the last element from each segment.

An implementation using the segmented scan functions can be seen below. `iota` n will create the array $[0, 1, \ldots, n-1]$ and the `reshape` expression will make a flat version of `xs` without performing a copy (the two arrays are aliased). The last line reads the last element from each segment.

```
fun segmented_reduce_sum(xs: [m][n]i32) : [m]i32 =
  let flags = map (\i -> i % n == 0) (iota (m*n))
  let xs_flat = reshape (m*n) xs
  let scanned = segmented_scan_sum (flags, xs_flat)
  in map (\i -> scanned[((i+1)*n)-1]) (iota m)
```

When I started my thesis, this was the strategy the Futhark compiler would use to implement a segmented reduction. This strategy has two major flaws, (1) it requires us to compute and store all intermediate results of the reduction, which is unnecessary; and (2) it does not exploit the fact that the input array is regular, which could allow for major simplification of the generated code.

Chapter 4

Achieving Good Performance on GPUs

In this chapter I will give a small introduction to GPU programming and summarize the most important factors for achieving good performance for GPU code. I will show how to implement reduction and transposition using CUDA (two important algorithms for the rest of my thesis), and I will show how transposing an array can help achieve a good memory access pattern (memory coalescing).

4.1 Basics of GPU Programming

A GPU has a number of *streaming multiprocessors* (SM) (OpenCL: compute unit) that each can run a large number of threads concurrently. A number of threads are bundled together in a *warp*: All threads within a warp executes the same instruction simultaneously.

In an `if-then-else` statement, if all threads within the same warp evaluate the condition to the same value only that branch will be executed; otherwise both branches will be executed, one after the other, and the threads will idle while the incorrect branch is being executed. The same logic applies to for-loops and the like.

The main idea of GPU programming is that we spawn a lot of threads to execute the same instructions on different parts of our input data. We do this by creating a single function, called a *kernel*, which all threads will execute. Threads are organized into *groups* (CUDA: blocks), and the groups are organized into a *grid*. All groups have the same size, which should be a multiple of the warp size. There is a maximum allowed group size so we must use multiple groups to work on large input. It is common practice to launch significantly more threads than can be active at once, called *oversubscribe* the hardware, because this lets the scheduler on the GPU start executing a new group of threads as soon as another group of threads have all finished executing their kernel.

To let each thread work on a different part of the input data, there are special variables/functions for a thread to get its index within a group, and

variables/functions to get the index of the group – a unique index can now be calculated as `thread_id = local_id + group_id * group_size`.

It is not possible for threads within different groups to synchronize their execution. If we want to perform multiple steps of computation with synchronization in-between, we can simply launch the first kernel and wait for it to finish before launching the second kernel.

Threads within the same group can exchange data, and have a fast mechanism to synchronize their execution. When launching a kernel, we can request an amount of fast on-chip memory only accessible by the threads in the same group, called *local memory* (CUDA: shared memory). Whenever we want to read data from local memory that has been written by a thread in a *different warp*, we need to synchronize all the threads in the group, because we do not know if the other warp has reached the point yet where the data would be written.

All threads have access to a large amount of slow off-chip *global memory* (CUDA: global memory). All NVIDIA GPUs above compute capability 2.x tries to speed global memory access up by employing a shared L2 cache, and some GPUs have a L1 cache per SM [8]. The global memory resides on the GPU device, so when we want to run a kernel on some data currently in the host memory (normal RAM of your node), we first need to transfer the data to the GPU; once the kernel is done we need to transfer the data back from the (GPU) device to the host.

When using a GPU to accelerate only parts of the computation of a program, before the GPU kernel can start, we will have to wait on transfering data from the host to the GPU device; After the GPU kernel is finished, we will also have to wait for the data to be transferred back to the host from the GPU device. In this thesis we are interested in making the most efficient implementation of the computation phase, so we will completely ignore this aspect. However, I will note that it is possible to apply pipelining to the process of transferring data and performing computation, which is called *streaming* in GPU terminology. Streaming can improve the total performance of accelerating parts of a program tremendously.

You can run multiple kernels at the same time on a GPU, but I do not use this in my thesis, so we will not go into details about this.

4.2 Important Factors for Good Performance

I will go over the most important factor for getting good performance from GPU code.

4.2.1 Memory Coalescing

Using the correct memory access pattern is the alpha and omega of GPU programming, because it has such a high impact on performance. In general terms the access pattern we are looking for is the following: If the threads in a warp is numbered $i, i+1, \ldots, i+k$, the threads should access elements $j, j+1, \ldots, j+k$

of the array respectively. This is called *memory coalescing*, and it also works if the threads access the same elements in a permuted fashion (e.g., in reverse order).

When accessing global memory in this way, we only need a single memory transaction to get the requested data for all k threads. I am skipping over some low level details to this, such as the fact that if the first element is not aligned to a 128-byte address, we might need two memory transactions; and that depending on whether the global memory is configured to be cached or not, the amount of data transferred in a single memory transaction changes; but overall this is the key insight.

Local memory (the fast-per group memory) is implemented using a number of equally sized memory modules, called *banks*, that can be accessed at the same time. I will consider the case of NVIDIA GPUs above compute capability 2.x, which has 32 banks, one for each thread in the warp. Successive data elements (32 or 64 bits) are stored in successive banks[1]. If all threads in a warp access data from a different bank, even if the access is not coalesced, all requests will be handled simultaneously. However, if multiple threads want to access different data elements from *the same bank*, the requests will be handled sequentially. So if thread i want to access data element $i \times 32$, the total time for memory access will be 32 times longer than if each thread was using its own bank. It is not a problem if multiple threads access the same data element, it will simply be broadcasted.

4.2.2 Occupancy

Once we are performing coalesced memory accesses, to get high performance we need to run enough threads to hide the latency of memory accesses. Usually we want to run as many threads as possible.

The *theoretical occupancy* of launching a kernel is defined as the number of active threads possible in this configuration, over the maximal number of active threads that can be run on the GPU.

$$\text{theoretical occupancy} = \frac{\text{possible active threads in this configuration}}{\text{maximal number of active threads on this GPU}}$$

The number of threads it is possible to run on a single SM is determined by a number of things:

1. There is a hard limit on the maximum number of threads, maximum allowed group size, and maximum number of groups per SM. Only launching one large group, or many small groups will not give high occupancy.

   As an example, on the GTX 780 Ti, a SM can have a total of 2048 threads, has a maximum group size of 1024, and the maximum number of groups is 16. So launching 16 groups with size 32, will only give us $\frac{16 \times 32}{2048} = \frac{512}{2048} = 25\%$ occupancy. Launching a single group with size 1024 will give us 50% occupancy, however we could just launch 2 groups of size 1024 to

[1] compute capability 3.x allows this to be configured to either 32-bit or 64-bit.

get to 100% occupancy. If we are using a group size of 768 we can at most fit 2 groups on a SM, and thus get $\frac{2\times768}{2048} = \frac{1536}{2048} = 75\%$ occupancy.

On the GTX 780 Ti, to get 100% occupancy with the smallest possible group size we need to use all 16 possible groups, so the group size must be $\frac{2048}{16} = 128$. Generally speaking group sizes of 128 and 256 are recommended because multiples of these can easily hit the maximum number of threads per SM within the limit of maximum number of groups per SM.

2. There is a hard limit on the number of registers, and on the amount of local memory per SM. If a kernel uses many registers per thread, or a large amount of local memory per thread, the limit of the maximum number of threads will come from these constraints – and therefore also what the best group size and number of groups is.

   On the GTX 780 Ti, there is 49152 bytes of local memory[2] and 65536 32-bit registers per SM. This means that to get 100% occupancy each thread can use at most $\frac{65536}{2048} = 32$ registers, and $\frac{49152}{2048} = 24$ bytes of local memory.

If you want to play around with different configurations on NVIDIA GPUs, I can recommend an occupancy calculator excel sheet NVIDIA have made: `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

Higher Performance with Lower Occupancy

For memory bound kernels it is possible to get higher performance with lower occupancy. This might seem like a contradiction, but the idea described in [31] is actually quite straightforward.

If all threads process multiple elements instead of one, they might be able to have multiple global memory transactions in flight at once. To store this data each thread might need more registers, which then forces the occupancy down. However, there might be a net benefit here; if we need to halve our group size, but each thread can process four elements in the same time as before, we have achieved a 2x speedup.

Generally speaking processing multiple elements per thread will lead to better performance for memory bound kernels, and will not always make the occupancy go down.

4.3 Example CUDA Programs

I will show a few examples of how to implement algorithms in CUDA, specifically how to perform a parallel reduction and how to perform a parallel transposition of an array. (For this section I will use the term *block* instead of *group*).

[2] This size is configurable, as 65KB of fast on-chip memory is divided between the L1 cache and the local memory. We use the configuration of 49152 bytes of local memory.

4.3.1 SAXPY – A Simple Example Program

To start off, we will look at a very simple example to get familiar with the code used to program CUDA. Below is a CUDA kernel for looping over all elements x, y of the arrays X, Y and computing $y = y + a * x$, with the constant a. Both of the arrays Y and X have size n. This computation is called single precision a times x plus y (SAXPY).

```
__global__ void saxpy(int n, float a, const float* X, float* Y)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  if (global_id < n) {
    Y[global_id] = Y[global_id] + a * X[global_id];
  }
}
```

The keyword `__global__` marks the function as a CUDA kernel callable from the host. A block is made up of a 3D arrangement of threads, and a grid is made up of a 3D arrangement of blocks; Most of the time we only use one dimension as in this example (transposition is an exception). The `blockIdx` has `x,y,z` fields for the index of the block in the grid, and `threadIdx` has `x,y,z` for the index of the thread inside the block. Likewise, `blockDim` has information on the shape of the block, and `gridDim` the shape of the grid.

We can compute a global index for a thread as `blockIdx.x * blockDim.x + threadIdx.x`. To fully process the array, we must launch $\lceil \frac{n}{blockDim.x} \rceil$ blocks.

```
float *h_X, *h_Y;
... // initialize h_X and h_Y

// copy data to device memory
float *d_X, *d_Y;
cudaMalloc(&d_X, n*sizeof(float));
cudaMalloc(&d_Y, n*sizeof(float));
cudaMemcpy(d_X, h_X, n*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_Y, h_Y, n*sizeof(float), cudaMemcpyHostToDevice);

// launch kernel
dim3 dimBlock(256, 1, 1);
dim3 dimGrid((n+256-1)/256, 1, 1);
saxpy <<<dimGrid, dimBlock>>> (n, a, d_X, d_Y);
```

The code above shows how to launch the `saxpy` kernel. We must first allocate memory on the GPU device for `X` and `Y`. Then we must copy the data `X` and `Y` from the host to the device. We must configure the shape of the blocks and the shape of the grid (we compute $\lceil \frac{a}{b} \rceil$ as $\frac{a+b-1}{b}$). Finally, we can launch the kernel with the special syntax `<<< grid , block >>>`.

4.3.2 Parallel Reduction

I will give an example for how to compute a sum-reduction in parallel using CUDA. I will only consider an implementation that can handle a non-commutative reduction operator. I will treat the simple case where each thread

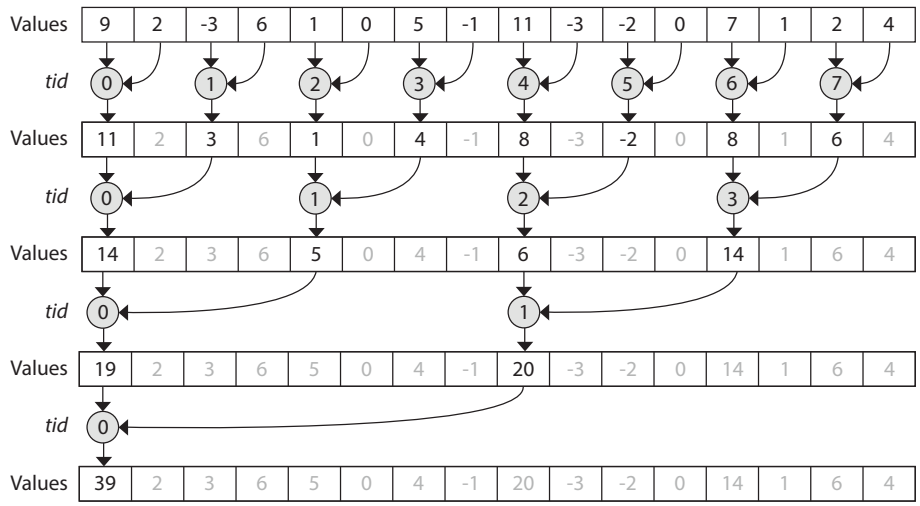


Figure 4.1: Computing reduction inside block using the first kernel version. Adapted from [15].

---

reads one element from the input array. Making each thread read multiple elements will improve performance: we will investigate this in subsection 5.2.2.

This example is partially based on [15], which gives a detailed explanation for how to compute an efficient commutative parallel reduction in CUDA. For simplicity, we will only handle cases where the size of the input data is a multiple of the block size, and the block size is a power of 2.

We will use multiple phases to compute the reduction. First we will launch multiple blocks to perform reductions on multiple separate parts of the array, and then we will reduce these intermediate results further until we have a single result. The main idea for computing a reduction within a block is to use a binary tree approach: at each level of the tree we will perform a reduction between the nodes are connected by the same parent.

We will explore two different implementations: the second implementation will improve on the performance of the first.

First version

The code for the first version of the kernel is listed below (and is adapted from [15]).

Each thread will put an element of the input array into shared memory. Then the tree-based reduction begins, which is illustrated in Figure 4.1. If we let s denote the size of a block, the iterations will proceed as:

- In the first iteration, the first $s/2$ threads will perform a reduction – thread with index tid will reduce element $tid \times 2$ and element $tid \times 2 + 1$, and store result at $tid \times 2$. Then we need to synchronize threads in the block, to

ensure that threads in different warps have all finished performing their reductions in this iteration before moving on to the next iteration.

- In the second iteration, the first $s/4$ threads will perform a reduction; thread tid will reduce element $tid \times 2$ and element $tid \times 2 + 1$, and store the result at $tid \times 2$. Again we need to synchronize threads here.
- We will continue iterating and halving the number of threads to perform a reduction, until there is only one thread left performing the last reduction.

```
__global__ void first_reduce_kernel(const float* idata, float*
    odata) {
    extern __shared__ float sdata[];
    const unsigned int tid = threadIdx.x;
    const unsigned int offset = blockDim.x * blockIdx.x + tid;

    sdata[tid] = idata[offset];
    __syncthreads();

    for (unsigned int s=1; s < blockDim.x; s*=2) {
        int index = 2 * s * tid;

        if (index < blockDim.x) {
            sdata[index] += sdata[index+s];
        }

        __syncthreads();
    }

    if (tid == 0) odata[blockIdx.x] = sdata[0];
}
```

If we look at the access pattern for the shared memory, we can see that in iteration with index i there is a stride of 2^i. So in the 5$^{\text{th}}$ iteration, we will use a stride of 32, and at most 32 threads[3]. This has the unfortunate consequence that all threads will use *the same bank* for their access to shared memory. If we are using a block size of 1024, we will get a 32-way bank conflict.

In the 6$^{\text{th}}$ iteration, we will use a stride of 64, and again all active threads (max 16) will use the same bank. It's not really looking good in the 4$^{\text{th}}$ iteration either to be honest, by using a stride of 16 there will be two 16-way bank conflicts in each warp.

If we could get rid of these bank conflicts, we should be able to get better performance, and that is what we will do in the second version.

Second version

This version of the reduction kernel uses the same overall idea: put one input element into the shared memory and then perform the reduction. However, the way of performing the reduction is significantly changed: We start by performing a reduction within each warp, then pass the intermediate values to the first warp which will reduce these values to get a final result.

[3]because of the max block size of 1024

Performing reductions within a warp has the benefits that there will be no bank conflicts, and we do not need to use `__syncthreads()`. As CUDA has a warp size of 32 and a maximum block size of $1024 = 32^2$, we will always be able to fit all the intermediate results of all warps in the first 32 elements of shared memory, and thus only use one warp for this step-2 reduction.

To handle the case where there are not 32 warps used in a block (i.e., block-size < 1024), after performing the reduction in warp 0, the threads in warp 0 will fill the shared memory at positions $[1, 31]$ with the neutral element (i.e., 0).

```
__device__ inline
void reduce_in_warp(volatile float* sh_mem, const unsigned int tid)
     {
    const unsigned int lane = tid & 31;

    // no synchronization needed inside a WARP
    if (lane %  2 == 0) sh_mem[tid] = sh_mem[tid] + sh_mem[tid+1] ;
    if (lane %  4 == 0) sh_mem[tid] = sh_mem[tid] + sh_mem[tid+2] ;
    if (lane %  8 == 0) sh_mem[tid] = sh_mem[tid] + sh_mem[tid+4] ;
    if (lane % 16 == 0) sh_mem[tid] = sh_mem[tid] + sh_mem[tid+8] ;
    if (lane % 32 == 0) sh_mem[tid] = sh_mem[tid] + sh_mem[tid+16];
}

__device__ inline
void reduce_in_block(volatile float* sh_mem, const unsigned int tid
    ) {
    const unsigned int lane   = tid &  31;
    const unsigned int warpid = tid >> 5;

    reduce_in_warp(sh_mem, tid);

    // fill blank values in position [1..31], if not 32 warps are
        used.
    if (warpid == 0 && lane != 0) sh_mem[tid] = 0;

    __syncthreads();

    // move intermediate result from this warp to the first warp.
    if (lane == 0) sh_mem[warpid] = sh_mem[tid];
    __syncthreads();

    if (warpid > 0) return;

    reduce_in_warp(sh_mem, tid);
}

__global__ void second_reduce_kernel(const float* idata, float*
    odata) {
    extern __shared__ float sh_mem[];
    const unsigned int tid = threadIdx.x;
    const unsigned int offset = blockDim.x * blockIdx.x + tid;

    sh_mem[tid] = idata[offset];

    reduce_in_block(sh_mem, tid);

    if (tid == 0) odata[blockIdx.x] = sh_mem[0];
}
```

I did not explore if using the threads of warp 0 to fill the intermediate array, thus generating a 31-way bank conflict, was faster than letting the first thread of

0	1	2
3	4	5
6	7	8
9	10	11

(a) Input array.

0	3	6	9
1	4	7	10
2	5	8	11

(b) Transposed array.

Figure 4.2: Example of transposing a [4][3] array into a [3][4] array.

each warp move its own intermediate result to the correct position in the shared memory.

There are two small notes concerning the code: (1) the keyword `__device__` marks a function as a GPU kernel callable from other GPU kernels; (2) normally the modulo expression `lane % x` would be expensive, but in this case the compiler can optimize it into `lane & (x-1)` because `x` is a power of 2.

Performance

I made a small performance comparison of the two versions, which can be found in appendix A. Version 2 gets a speedup of more than 2x over version 1 when using a block size of 1024, and is never slower than version 1 – though the difference is very small when using a block size of 128. The fully optimized commutative reduction kernel from NVIDIA has better performance than version 2, and in one case achieves a speedup of 2x over version 2.

4.3.3 Transposition

Transposition is an important GPU kernel because it can help us achieve memory coalescing as we shall explore in section 4.4.

If we have a 2D array of size $[height][width]$, we can transpose this into a $[width][height]$ array. For the purpose of illustration, Figure 4.2 shows the result of transposing a $[4][3]$ array into a $[3][4]$ array.

To understand the implementation strategy we will go through a small example. Assume we use a block shape of $(x = 32, y = 32, z = 1)$, and that we have a 2D array of size $[32][64]$ we would like to transpose into a $[64][32]$ array. We can do this by launching 2 groups, each will read all elements of a 32x32 section of the input array, and write them to the correct position in the output array. We will call the square section of the arrays a *tile*, and the number of threads in each direction the *tile size*.

The code shown below is an optimized version of a transposition kernel using this strategy. It uses shared memory to store the result of reading the tile, which allows each threads to write a different element than it read – otherwise writing from threads in the same warp would use a stride global memory access, which is not good. The code also used a slightly larger amount of shared memory than needed ($[TILE][TILE + 1]$ instead of $[TILE][TILE]$) to avoid

that all threads in a warp, that have the same index in the y-dimension iy but different indexes in the x-dimension ix, should access shared memory at $[ix][iy]$ which would be served by the same bank (bank iy).

More details for how to arrive at this implementation can be found in [14]. The code shown here represents the transposition code Futhark uses, and differ slightly from what is presented in [14]: this version does not process multiple elements per thread, which can yield an improvement in performance.

```
// blockDim.y = TILE; blockDim.x = TILE
// each block transposes a square (TILE x TILE) part of the input
    array A
template <int TILE>
__global__ void transpose_futhark(const float* A, float* B, int
    heightA, int widthA) {

    __shared__ float tile[TILE][TILE+1];

    int x = blockIdx.x * TILE + threadIdx.x;
    int y = blockIdx.y * TILE + threadIdx.y;

    int ind = y * widthA + x;
    if (x < widthA && y < heightA) {
        tile[threadIdx.y][threadIdx.x] = A[ind];
    }
    __syncthreads();

    x = blockIdx.y * TILE + threadIdx.x;
    y = blockIdx.x * TILE + threadIdx.y;

    ind = y * heightA + x;
    if (x < heightA && y < widthA) {
        B[ind] = tile[threadIdx.x][threadIdx.y];
    }
}
```

Futhark Implementation

Futhark currently uses a transposition with tile size of 16. This has the unfortunate effect that on NVIDIA GPUs that has warp size of 32 (and 32 banks), we will run into 2-way bank conflicts: assuming that `tile[0][0]` is served by bank 0, thread 16 within the first warp will access `tile[1][0]` which is served by bank 17 (because bank 16 should serve the element at `tile[0][16]`). By the same logic, thread 31 will access the element at `tile[1][15]` which will be served by bank 0. So we have a 2-way bank conflict, because two threads want to access different elements from the same bank (This will indeed happen for every warp, and not just the first one).

This problem will not occur when using TILE size of 32, as all threads in a warp will access their own bank. I will show the difference in performance in subsection 5.2.4.

0	1	2	3	4	5	6	7	8	9	10	11

(a) Input array.

0	1	2
3	4	5
6	7	8
9	10	11

(b) Input array as [4][3].

0	3	6	9
1	4	7	10
2	5	8	11

(c) Transposed array.

Figure 4.3: Transposition can allow memory coalescing when using non-commutative reduction.

4.4 Transposing to Achieve Memory Coalescing

We are in trouble if we want each thread in a kernel to read multiple elements from an array in consecutive order, because this access will not be coalesced.

If there are t threads that all want to read c elements of a 1D input array of size $t \times c$, we can make the array accesses coalesced by this trick: We will pretend the 1D input array is a 2D array with the shape $[t][c]$, to transpose the array, which will get the shape $[c][t]$. Now the threads will access the global memory in a coalesced way, as a thread with global id gi will find its elements at indexes

$$gi\ ,\ (gi + t)\ ,\ (gi + t \times 2)\ ,\ \ldots\ ,\ \big(gi + t \times (c - 1)\big)$$

Figure 4.3 shows an example of how this works. We are using an input array of size 12, we will use a total of 4 thread ($t = 4$) and each thread should read 3 elements ($c = 3$). The "pretend" array of size $[4][3]$ can be seen in Figure 4.3b, and the result of transposing can be seen in Figure 4.3c. Now the 4 threads can read one element from the same row in each iteration, so the access is coalesced.

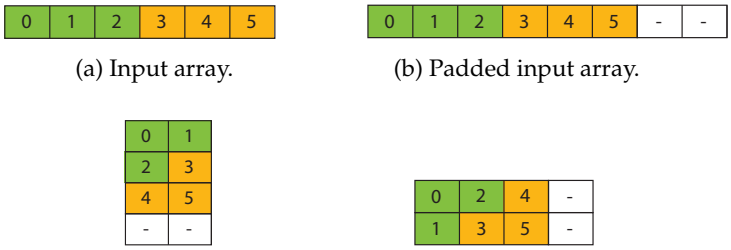
If the size of the input array is not a multiple of the total number of threads t, we cannot transpose it. To handle this case, we extend the array so it becomes a multiple of t, which we can then transpose.

4.4.1 Segmented Input

We can also transpose a segmented array, to allow for coalesced access. We use the same approach as before, and just need a more complex indexing function.

In Figure 4.4 we transpose a [2][3] array to be used by 2 groups of 2 threads each ($t = 4$) and each thread should read 2 elements ($c = 2$). After padding the array to length $t \times c = 8$, we can treat it as a [4][3] array and transpose it to a [3][4] array.

When reading the first element, the indexing function must make sure that thread 0 and 1 in group 0 read from index 0 and 1 respectively; and must make sure that thread 0 and 1 in group 1 read from index 6 and 7.



(a) Input array. (b) Padded input array.

(c) Padded array as [4][2]. (d) Transposed padded array.

Figure 4.4: Transposition of 2 segments with size 3, when wanting to use 2 groups with 2 threads, and thus a chunking=2.

4.5 Performance of Memory Bound Kernels

The performance of kernels that perform very little computation per element is bounded by the memory bandwidthcuda-transpose of the GPU device, this is what we call a *memory bound* kernel. It is often instructive to measure the performance of a memory bound kernel as the memory bandwidth it utilizes as a percentage of the maximum bandwidth. The performance of a good implementation for reduction with a low-intensity reduction operator, such as addition, will be bounded by memory.

We can compute the theoretical maximum bandwidth (also called the peak bandwidth) of a GPU device by using the following formula with its memory frequency and the memory bus width in bytes:

$$\text{peak bandwidth in GiB/s} = \frac{\text{memory freq} \times \text{bus width} \times 2}{1024^3} \tag{4.1}$$

where the $\times 2$ comes from the fact that the bus is running double data rate (DDR).

However, it is often not possible to reach 100% utilization of the available bandwidth, even for performing a memory copy. When using a small dataset it is also not possible to utilize the hardware fully, so measuring bandwidth of a kernel that only processes an array of 100 elements will not give any meaningful data.

The GTX 780 Ti has a memory clock rate of $3500\,\text{MHz}$ and memory bus width of 384 bits. This yields a theoretical memory bandwidth of

$$\frac{3500 \times 10^6 \times \frac{384}{8} \times 2}{1024^3} = 312.92\,\text{GiB/s}$$

for completeness, in gigabytes per second, the bandwidth is 336 GB/s.

Chapter 5

Prototype

My idea for computing segmented reductions efficiently involved using three different strategies to effectively cover all cases of number of segments and segment size:

- When there are so many segments that we can fully utilize the GPU by computing the reduction for each segment sequentially in a single thread, my plan was to do so.
- When there are few large segments I planned to use an approach similar to a one-dimensional reduction, where we use multiple groups to perform the reduction over a single segment.
- when there are too few segments that using the sequential approach is not efficient, and the segment sizes are so small that the first approach will not be efficient, my plan was to use a single group to reduce multiple segments.

In this chapter we will explore these three strategies, and their performance when computing a segmented sum. We will use a fixed number of total elements, and study the performance on different configurations of number of segments and segment size, ranging from a single segment that contains all the elements, to the other extreme of all the segments only containing a single element.

Initially I will give some more details on each strategy and their implementation (section 5.1). Then we will look at how the maximum memory of the GPU will determine the best possible runtime for a given configuration of number of segments and segment size (subsection 5.2.1).

We will continue by examining how making each thread of a group read multiple elements can significantly improve performance (subsection 5.2.2), and will study the raw performance of each of the three strategies (subsection 5.2.3).

Transposing the input array can be required to enable coalesced memory accesses in the final computation of segmented reductions, so we will study the cost of performing a transpose (subsection 5.2.4). We will see that when

either the width or the height of the input array is low, the normal transpose kernel is very inefficient; I will briefly introduce a new transpose kernel that will improve performance in those cases.

After having looked at the these performance measurements, I will present a simple algorithm to decide which strategy to use for a given configuration of number of segments and segment size (section 5.3). We will look at the final performance of computing a commutative and a non-commutative segmented reduction using all three implementation strategies; we will use this data to get an idea of how good predictions where made by the decision algorithm, and discuss ways to allow an autotuning project to influence the decision algorithm.

Limitations

I chose to focus solely on segmented reductions, and not segmented redomaps, to allow me to getting this part of the equation right. I will also hypothesize that a good approach for computing segmented reductions will be applicable to segmented redomaps. We will see if this hold when when studying the performance of my implementation in Futhark in chapter 6.

Furthermore, I narrowed the scope of this part by only using sum as the reduction operator. My intuition is that using such a low intensity reduction operator will allow us to focus on the cost of the algorithmic approach of computing a segmented reduction, that is, we can see much of the maximum bandwidth an implementation is capable of utilizing. I will hypothesize that a good approach for segmented reduction with low intensity reduction operators will also be applicable when using high intensity reduction operator.

Sum is a commutative reduction operator, but we will use it in the non-commutative case as well, to be able to compare performance between commutative and non-commutative reductions.

I will not try to manually tune my implementation to give the best runtime, instead my goal is to develop a general approach that is applicable to any problem on any GPU.

5.1 Overview & Implementation of Kernels

In this section I will give some more detail to each of the three strategies for computing a segmented reduction outlined above. I will also try to convey the an understanding for how each strategy can be implemented as a GPU kernel.

The idea behind investigating all three strategies is that we can choose between them to optimize performance, for a given problem and a configuration of number of segments and segment size.

5.1.1 Loop-in-map

The first strategy is using a single thread to sequentially compute the reduction of a whole segment; as this can be expressed in Futhark as a loop within a map,

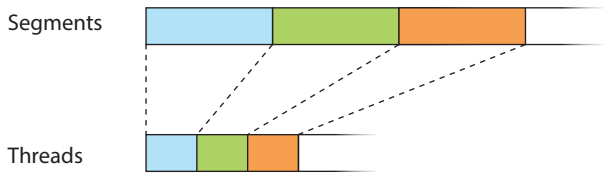


Figure 5.1: The loop-in-map kernel will process each segment sequentially in one thread.

```
__global__ void loop_in_map (...) {
    segment_index = group_id * group_size + local_id;
    acc = neutral_element;
    for (int i = 0; i < segment_size; i++) {
        elem = input_array[segment_index + i*num_segments];
        acc = reduction_operator (acc, elem);
    }
    output_array[segment_index] = acc;
}
```

Listing 5.1: Pseudo-CUDA-code for implementing the loop-in-map kernel

we will call this strategy *loop-in-map*. This strategy is illustrated in Figure 5.1, where we can see each of the three segments being processed by their own thread.

My initial idea was that this strategy should only be used when there are enough segments that we can fully utilize the hardware by using a single thread per segment.

Using loop-in-map will require us to transpose the input array to achieve memory coalescing as described in section 4.4.

Implementation

The loop-in-map approach has a very straightforward implementation, which can be seen in the pseudo-CUDA-code in Listing 5.1. We must launch at least as many threads as there are segments. Each thread will go over all the elements of a segment and apply the reduction operator.

To achieve memory coalescing we must transpose the input array; therefore, a thread will read its elements starting at the index of the segment and use a stride equal to the number of segments.

5.1.2 Large

If the segment size is larger than the group size, we can use multiple groups to reduce one segment. This will allow us to use more active threads, than just

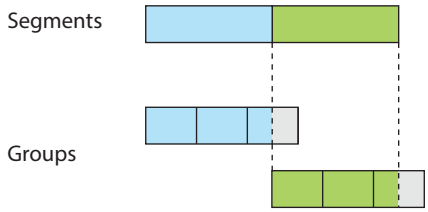


Figure 5.2: The larger kernel can use multiple groups to reduce a single segment; this will generate multiple intermediate results that will need to be reduced further. In this example we use three groups for each of the two segments.

---

using a single group per segment. When there is only a few large segments, this will give us much better performance. We will call this strategy *large*, as it is used to process large segments.

We do however end up with a number of intermediate results per segment that will have to be reduced afterwards; this second reduction will be over orders of magnitude fewer elements, so should be extremely fast.

Figure 5.2 shows an example of using the large kernel to reduce two segments, using three groups per segment. The segment size is not a multiple of the group size, so some threads in the last group for each segment is wasted because there is not any elements for them to read. In Figure 5.2 we will get 3 results per segment that will need to be reduced further to get a single value per segment.

When the segments are very large, instead of launching thousands of groups per segment, we can make each thread of a group read multiple elements. This can improve performance significantly, because multiple memory transactions can be in flight at once (as described in subsection 4.2.2). This will also have the added benefit of reducing the number of intermediate results produced per segment. For a non-commutative reduction, making each thread multiple elements will require that the input array has been transposed, to achieve memory coalescing (as described in section 4.4).

If there are so many segments that it would not increase occupancy to use multiple groups per segment, we can use a single group where each thread reads multiple elements to reduce an entire segment, thereby also avoiding the cost of the recursive reduction of intermediate values.

The large kernel cannot be used to reduce more than one segment per group. When the segment size is much lower than the group size, we will only be able to use a small fraction of the threads in a group with the larger kernel, so we cannot expect it to be very efficient.

Implementation

The larger kernel can be implemented very similarly to the second parallel reduction kernel from subsection 4.3.2.

```
__global__ void large (...) {
    segment_index = group_id / num_blocks_per_segment
    elem_inc = is_commutative ? threads_in_segment : 1
    stride = is_commutative ? threads_in_segment : <...>
    elem_idx = <...>
    offset = <...>

    acc = neutral_element;
    for (int i = 0; i < chunking && elem_idx < segment_size;
         i++, elem_idx += elem_inc) {
        elem = input_array[offset + i*stride];
        acc = reduction_operator (acc, elem);
    }

    sh_mem[tid] = acc;
    reudce_in_block(sh_mem, tid);
    if (tid == 0) output_array[blockIdx.x] = sh_mem[0];
}
```

Listing 5.2: Pseudo-CUDA-code for implementing the large kernel

Pseudo-CUDA-code for implementing the large kernel is in Listing 5.2. Compared to subsection 4.3.2 we have added a loop that will read at most `chunking` elements, or stop if the element it is about to read is out of bounds; for example, if the group size is 128 and the segment size is 255, we will use `chunking=2` but we should make sure the last thread only reads a single element.

Notice there is a difference between using a commutative and non-commutative reduction. To use a non-commutative reduction with chunking> 1 we must transpose the input array to get memory coalescing, and therefore the `offset` and `stride` calculation for indexing the input array will be different than for a commutative reduction. For a commutative reduction, we will use a stride equal to the total number of threads working on a segment.

It is important to understand that the threads in a non-commutative reduction will read consecutive elements of the *original input array*. This means that the way a $groupsize + 1$ sized segment will be handled is different: the threads in a commutative reduction will read one element each, except for the first thread that will read two elements; whereas for a non-commutative reduction, the first half of the threads will read two elements each, and then the next thread will read the last element.

5.1.3 Small

If the segment size is smaller than the group size, we might be able to process multiple segments inside a single group. This will be more efficient than using one group of the same size to process one segment, as the percentage of threads that can would be able to read an element must be low; that is

$$\text{percent active threads} = \frac{\text{segment size}}{\text{group size}} \leq 50\%$$

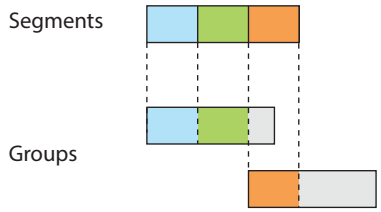


Figure 5.3: The small kernel can process multiple whole segments within a single group. In this example each group can process two whole segments, but there is only one segment to reduce for the last group.

---

we will call this strategy *small*, because it handles small segment sizes.

For simplicity, we will only consider processing a whole number of segments within one group, so we do not need to transfer intermediate results between groups. We might waste some threads because we cannot fill the group completely.

Figure 5.3 shows an example of using the small kernel; we can fit 2 whole segments within a single group, so we need 2 groups to process the three segments. The last group will have more "wasted" threads because there is only one segment for it to reduce.

Input data will not need to be transposed, because each thread will only read one element.

Implementation

The small kernel can be implemented using a segmented scan within the group/block (see subsection 3.4.1 for details on segmented scan).

The pseudo-CUDA-code in Listing 5.3 illustrates how the small kernel can be implemented. To handle spare threads that should not read elements of a segment, we compute a boolean flag `isactive`. A wasted thread needs to participate in the segmented scan, so we just use the neutral element as the element value. To compute the segmented scan we will modify the reduction operator to use a flag array as we did in subsection 3.4.1.

An important difference between performing a normal segmented scan and a segmented scan *within* a group is, that we can create the flag array in the fast local memory (this is faster than reading the elements from global memory); Furthermore, we only need to write some of the results from the in-group-segmented-scan to global memory, compared to all the results for a normal segmented scan. These differences should make the small kernel much faster than performing a general segmented scan.

I took the approach of a segmented scan because it was simple. I believe that it would be possible to create a more efficient solution to the problem of small segment sizes, but I did not choose to invest my time here.

```
__global__ void small (...) {
    isactive = <...>

    if (isactive) {
        segment_index = <...>
        offset = <...>
        elem = input_array[offset]
    } else {
        elem = neutral_element;
    }

    sh_mem_elem[tid] = elem;

    flag = tid % segment_size == 0;
    sh_mem_flag[tid] = flag;

    segscan_in_block(sh_mem, sh_mem_flag, tid)

    if (isactive && tid < segments_per_group)
        output_array[segment_index] = sh_mem[(tid+1)*segment_size];
}
```

Listing 5.3: Pseudo-CUDA-code for implementing the small kernel

5.2 Performance Experiments

Now we will look at the performance for the three different approaches. How much will reading multiple elements per thread improve the performance of the large kernel? What is the performance of the three kernels on different input data? And what is the cost of performing a transpose?

The experiments are run on a GTX 780 Ti. Reported runtimes are averages of running a kernel n consecutive times[1], after performing an initial warmup run.

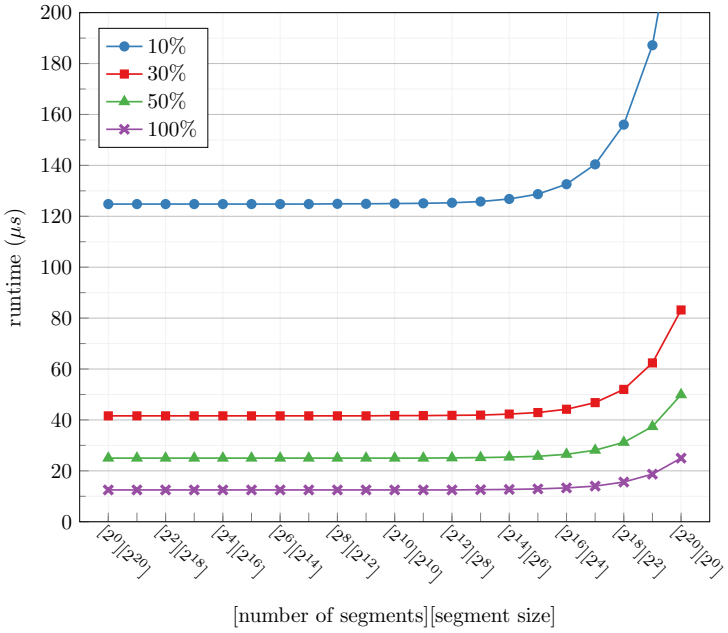
I will use row-major array notation for describing a configuration of number of segments and segment size: $[m][n]$ will denote an array with m segments that each have size n.

I will mostly use two sizes for the total number of elements: $2^{20} = 1\,048\,576$ and $2^{26} = 67\,108\,864$. I only use 32-bit floating points as the element type. 2^{20} 32-bit floating points is $4\,\mathrm{MiB}$ of data, and 2^{26} 32-bit floating points is $256\,\mathrm{MiB}$ of data.

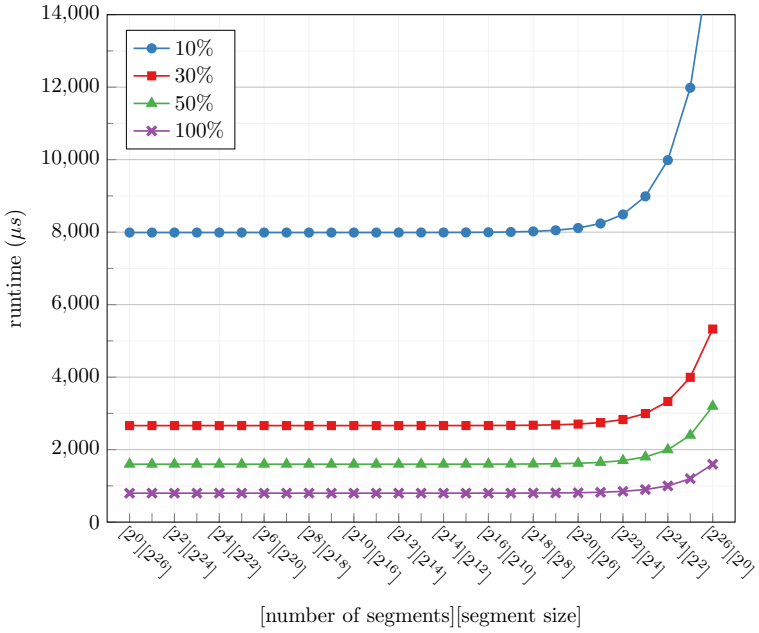
Terminology

When a thread reads multiple elements, we will call this *chunking*, and we will call the number of elements read per thread the *chunking factor*. We will sometimes write that we use chunking=n, to mean that we use a chunking factor of n.

[1] This will not launch the kernels concurrently. To do this we must launch the kernels in different streams.



(a) Using dataset of 2^{20} 32-bit floating points.



(b) Using dataset of 2^{26} 32-bit floating points.

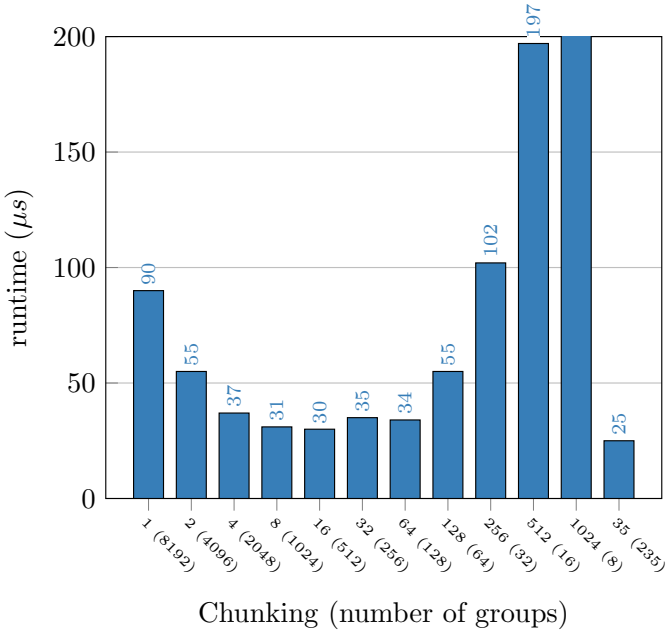
Figure 5.4: Time it will take to perform a segmented reduction on different configurations of number of segments and segment size, when utilizing 100%, 50%, 30%, and 10% of peak bandwidth on a GTX 780 Ti.

5.2.1 Bandwidth as The Limiting Factor

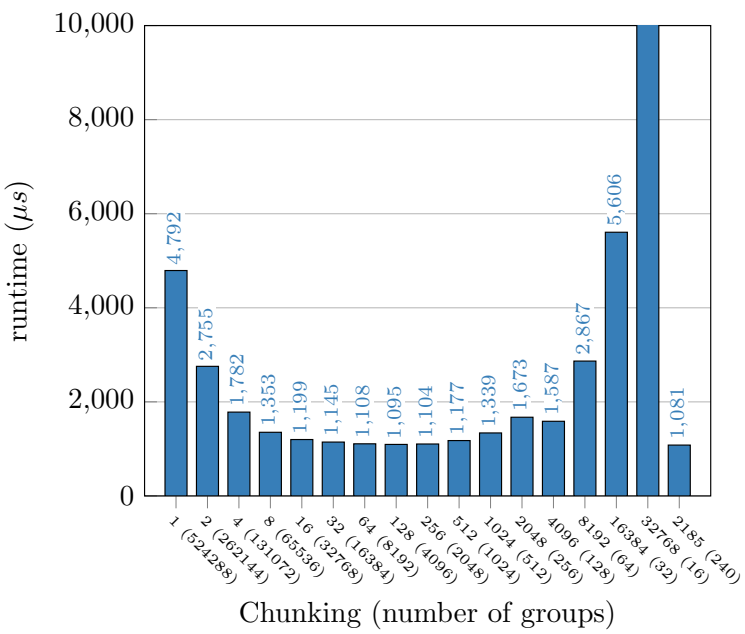
In section 4.5 we discussed how the performance of a kernel that does very little computation per element, would be bounded by the peak memory bandwidth of the GPU device. Before we begin looking at experiments for segmented reductions, we will study what performance to expect given a configuration of number of segments and segment size. We will ignore any reads and writes to temporary arrays that would occur in the multi-step approach, as they are "incidental". So we are looking at the best case possible performance.

If we are given a single segment with a size of 2^{26} we need to read all elements of the array, but only need to write one result: in total $(2^{26} + 1)$ elements read and written to memory. If we are given 2^{26} segments all with a single element, we still need to read all elements of the array, but also to write 2^{26} results: in total $(2^{26} + 2^{26})$ elements read and written to memory. Therefore, we can conclude that an *optimal* solution to computing segmented reductions that utilizes 100% of peak bandwidth, will be nearly twice as fast for the first configuration.

Figure 5.4 shows the time it will take to perform a segmented reduction on different configurations of number of segments and segment size, when utilizing 100%, 50%, 30%, and 10% of peak bandwidth on a GTX 780 Ti. The peak bandwidth for the GTX 780 Ti is 312.92 GiB/s, so 50% is 156.46 GiB/s, 30% is 93.88 GiB/s, and 10% is 31.29 GiB/s. Figure 5.4a shows the results when using a total of 2^{20} 32-bit floating points, and Figure 5.4b shows the results when using a total of 2^{26} 32-bit floating points. In both cases the optimal runtime is almost constant for the first many configurations, but starts increasing rapidly towards the end. The last configuration where there is only a single element in each segment, will take twice as long to process as a single segment containing all elements.



(a) Using dataset of $[1][2^{20}]$ 32-bit floating points.



(b) Using dataset of $[1][2^{26}]$ 32-bit floating points.

Figure 5.5: The performance impact of increasing the chunking factor. We use a constant group size of 128 to process a single large segment. We vary the chunking factor, and therefore also the number of groups.

5.2.2 The Importance of Chunking

Here I will try to answer the question of how large the benefit of chunking, and give a method of calculate a reasonable chunking factor to use.

I created an experiment, processing a single large segment of 2^{20} or 2^{26} elements using a constant group size of 128. We will use the chunking factors $1, 2, 2^2, 2^3, \ldots$, and therefore also vary the number of groups used.

We only look at the first step of performing the overall segmented reduction, so the result will be an array with one intermediate result from each group. This allows us to focus on the optimal chunking value.

The results from running this experiment can be seen in Figure 5.5 (results are averaged over 5 consecutive runs). We can see that we can get a great improvement in performance by using a chunking factor greater than 1, but how should we determine what chunking factor to use *before* looking at a figure like this?

As explained earlier, the maximum number of concurrently executing threads on the GTX 780 Ti is 30720. By using a group size of 128, we need at least $\frac{30720}{128} = 240$ groups to fully utilize the hardware. Therefore, I added a chunking factor such that we would use 240 groups (chunking of 35 for the 2^{20} case, and 2185 for the 2^{26} case). In both cases using 240 groups gives the best performance of all the tested combinations. In Figure 5.5b we can see that launching many more groups (e.g., 4095), while still using a fairly high chunking factor, achieves almost the same performance.

Performance drops off significantly after using 64 or less groups for both cases. This is because we are using a low amount of total threads ($\ll 30720$), so the hardware is not being fully utilized.

We do see that performance drops a bit when using 256 groups in both Figure 5.5a and Figure 5.5b. How can that be? For the purpose of simplicity let us assume that the scheduler can start 240 groups of size 128 at the same time, and that a group spends $chunking$ time units to finish its execution. This means that all 240 groups will finish at the same time, and a new batch of groups can start. In the case of 256 groups, we will have to launch two batches of groups, with 240 groups in the first, and 16 in the next. This means we will spend a total of $chunking \times 2 = 2048 \times 2 = 4096$ units of time. In the case of 4096 groups, we will have to launch $\lceil \frac{4096}{240} \rceil = 18$ batches, taking a total of $chunking \times 18 = 128 \times 18 = 2304$ units of time. This is obviously an oversimplification, but I think it is a reasonable explanation for why using 256 groups gives slightly worse performance.

So why isn't the performance of using 128 groups as bad? Well, we can probably attribute that to the fact that higher chunking is better, and that there is probably some kind of benefit of scheduling fewer groups.

The overall lesson learned here is that using chunking can improve performance tremendously, and that we should aim for 100% occupancy under the whole kernel execution.

Calculating Chunking Factor

In the following, when using a group size of 128, I will use this simple equation for calculating the number of groups to use per segment:

$$\text{number of groups per segment} = \left\lceil \frac{240}{\text{number of segments}} \right\rceil \tag{5.1}$$

for example, using 2 segments will give us 120 groups per segment, and using 64 segments will give us 4 groups per segment.

We will use this equation for calculating the chunking factor

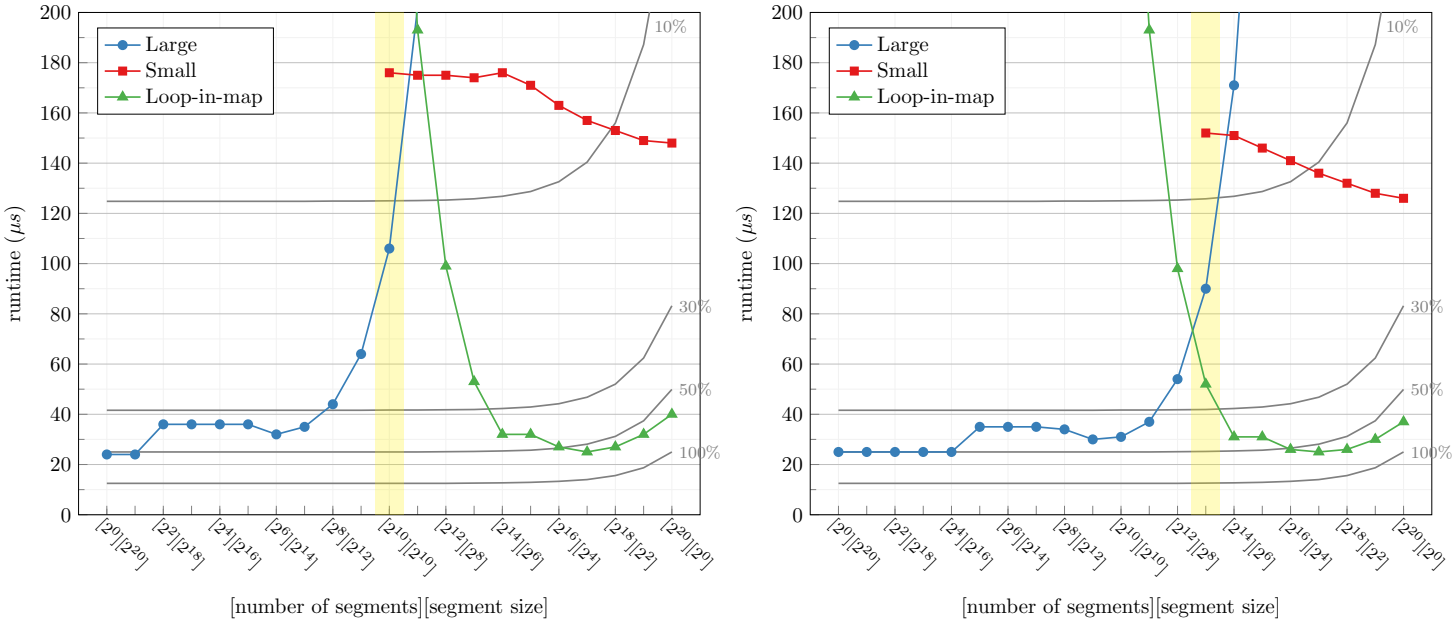
$$\text{chunking factor} = \left\lceil \frac{\text{segment size}}{\text{number of groups per segment} \times 128} \right\rceil \tag{5.2}$$

for example, using a segment size of 500 and 256 threads in total per segment will give us a chunking factor of 2.

If I use a different group size, we can use the same equations with adjusted constants.

5.2.3 Raw Performance Comparison

Now we will compare the performance of the three kernels, using different configurations of number of segments and segment size. We will only look at the *raw* performance of the kernels, so we will only look at the first step of performing the overall segmented reduction, and ignore transpositions; after looking at the performance of transpose in the next section, we will look at the big picture for both commutative and non-commutative segmented reductions.



(a) Using group size 1024.

(b) Using group size 128.

Figure 5.6: Performance on 2^{20} 32-bit floating points when using the large, small, and loop-in-map kernels in different configurations of number of segment and segment size.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size (chunking=1). The gray lines mark the runtime if utilizing the peak bandwidth by that percentage.

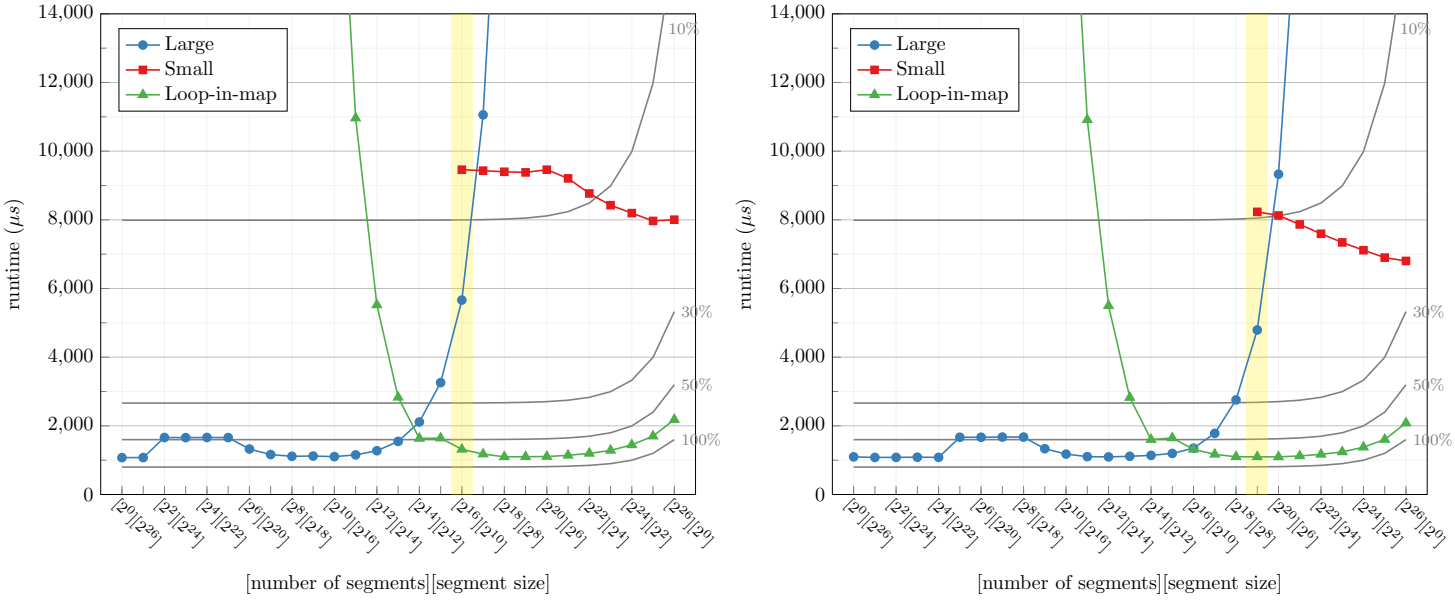
Figure 5.6 shows the performance of the three kernels, where the x-axis corresponds to using different configurations of 2^{20} 32-bit floating points. In Figure 5.6a we have used a group size of 1024, and in Figure 5.6b we have used a group size of 128.

Figure 5.7 shows the performance of the three kernels, where the x-axis corresponds to using different configurations of 2^{26} 32-bit floating points. In Figure 5.7a we have used a group size of 1024, and in Figure 5.7b we have used a group size of 128.

In all figures the yellow vertical bar marks the configuration where the number of segments is equal to the group size (chunking=1). The gray lines mark the runtime if utilizing the peak bandwidth by that percentage. Reported times are averaged over 10 consecutive runs.

We can make the following interesting observations:

- The group size has a large impact on how long the large kernel is viable to use. Using a group size of 128 allows us to get good performance for a larger portion of the input space, because we can postpone the problem of only being able to use a small fraction of the threads in a group.
- The loop-in-map kernel increases in runtime as the number of segments becomes very high. As we can see from the gray bandwidth-percentage lines, this is due to the fact that more results have to be written to memory.
- Performance of the small kernel increases significantly as the segment size becomes smaller. This happens even though there are more results



(a) Using group size 1024.

(b) Using group size 128.

Figure 5.7: Performance on 2^{26} 32-bit floating points when using the large, small, and loop-in-map kernels in different configurations of number of segment and segment size.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size (chunking=1). The gray lines mark the runtime if utilizing the peak bandwidth by that percentage.

---

to write to global memory, and we are using the same number of threads and groups.

As we can see from the gray bandwidth-percentage lines, the small kernel does not even utilize the bandwidth by 30% of its peak; so the small kernel is bounded by the computations it has to perform, and not by the memory bandwidth.

Therefore, a reasonable explanation for the performance increase is that when there are fewer elements in a segment, the segmented scan will perform fewer reductions between elements (it will only check the flag arrays).

- The performance small kernel seems really bad when compared to the good cases for the large and the loop-in-map kernel. However, this is an unfair comparison. The loop-in-map kernel will need a transpose, so the results here simply show that it has high throughput, and not necessarily that is will have better runtime than the small kernel in the end.

  For the large kernel, we should also compare to the case where chunking=1, simply because the small kernel does not use a chunking. We can see that in all cases, the runtime of the small kernel is within a factor 2 of the runtime of the large kernel with chunking = 1.

- There is a performance penalty of using a group size of 1024 over a group size of 128. We can see this in the cases where large uses chunking=1 (i.e., the point on the x-axis marked by the yellow bar), using a group size of 128 has a significant advantage: for 2^{26} elements using group size of 1024 with chunking=1 takes $5662\,\mu s$, and using group size of 128 with

chunking=1 takes $4791\,\mu s$. This can also be seen in the "last" case for the small kernel, where the segment size is 1.

This phenomenon can be explain by the fact that synchronizing 1024 threads compared to 128 threads have a significant higher cost. Each thread will have to wait longer for all other threads to reach the synchronization point, causing lower throughput.

- For the large kernel, the performance takes a small hit after we start using more than a few segments. For the group size 1024 this happens when using $2^2 = 4$ or more segments, and 128 this happens when using $2^5 = 32$ or more segments. The reason is that this is the point where we go from using the "optimal" number of groups, to a slightly larger number of groups. I described this effect in subsection 5.2.2 for the case of group size 128, where we saw the same performance degradation when using 256 groups instead of 240 groups.

  For both group sizes 1024 and 128, and for both cases of number of total elements, we see four entries with worse performance, before the performance improves again. For these four cases the large kernel will use the same slightly-more-than-optimal-number-of-groups, and then it will start to use a number of groups that are equal to the number of segments (which is then larger than this slightly-more-than-optimal-number-of-groups).

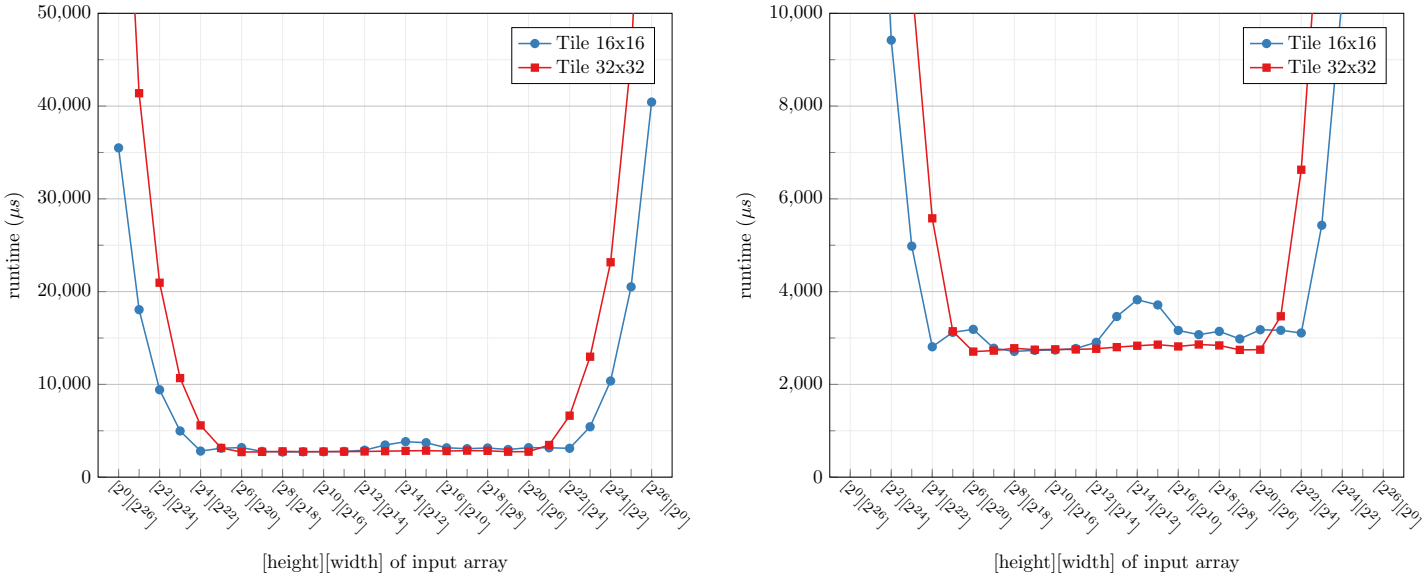
- When comparing the performance against the gray bandwidth-percentage lines in Figure 5.6 and Figure 5.7, we can see that the amount of data used has a big impact on the bandwidth we are able to utilize. In Figure 5.6 we only utilize more than 50% of the peak bandwidth by the good cases of the loop-in-map kernel, whereas in Figure 5.7 we are able to utilize more than 50% of the bandwidth for every configuration of number of segments and segment size, except for the ones where the large kernel uses the "bad" number of groups.

I also ran this experiment with group sizes 512 and 256, but the only interesting observation I made is that using group size 256 has no performance penalty over using group size of 128, there is only a difference in when the kernels are viable to use. Plots for both input sizes with all four group sizes can be found in appendix B.

Using a group size of 64 would not help extend the number of configurations that we could use the large kernel on, at least not on the GTX 780 Ti; With the group size 128, we are using as many concurrent groups as possible on the GPU, so by using group size of 64 we could only get 50% occupancy! See subsection 4.2.2 for more details.

5.2.4 Transpose Cost

We need to transpose the input array when using the loop-in-map kernel and when computing a non-commutative reduction using the large kernel with a chunking factor greater than one. We do this to get memory coalesced access



(a) Overview

(b) Close up

Figure 5.8: Performance of transposing an array. We use both groups of 1024 threads to transpose a 32×32 slice of the input array each, and groups of 256 threads to transpose a 16×16 slice each.

---

to global memory. Therefore, it is interesting to evaluate how expensive a transpose is.

Recall from subsection 4.3.3 that when using a total of t threads that should read (at most) c elements, we will pad the array to have length $t \times c$ if needed, and "pretend" it is a $[t][c]$ array to transpose it to a $[c][t]$ array.

We will only look at arrays that does not need to be padded, as a more specialized transpose kernel could remove the need for performing this padding step (i.e., by taking a `max_index` argument).

In subsection 4.3.3 I mentioned that Futhark uses a tile size of 16 causing a 2-way bank conflict, and that this would not occur if using a tile size of 32. Figure 5.8 shows the result of performing a transpose using these two kernels, over the usual 2^{26} 32-bit floating points. Reported times are averaged over 10 consecutive runs.

We can see that using a small size for either dimension in the transpose yields horrible results. This is because the transpose kernel will always create groups with shape $[tilesize][tilesize]$, but only threads with indexes that is not out-of-bounds will perform any work. In the case of transposing a $[1][n]$ array, only $\frac{1}{tilesize}$ of the threads in a group will perform any work. This could hint at why a 16x16 transpose kernel was chosen over a 32x32 kernel, as the 16x16 kernel handles these edge-cases a bit better[2]. However, in Figure 5.8b we do that using 16x16 for the tile size causes a bit of loss in performance around $[2^{14}][2^{12}]$, which does not occur when using a tile size of 32x32. I have not been able to explain this. Maybe it is the 2-way bank conflict showing up, although that is just a shot in the wind.

The bad performance when the width of the input array is low, is very troublesome. This means that if our chunking factor is low (< tile size), a transposition will be very costly.

[2] or it could simply stem from the time when there were only 16 banks on the GPU.

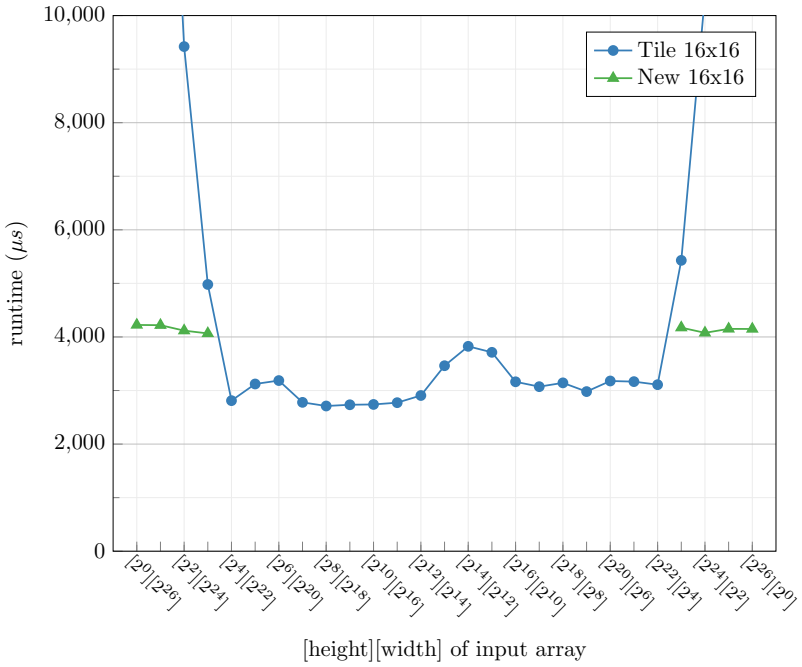


Figure 5.9: The performance of the new transpose kernel

When computing a non-commutative reduction, we could avoid this problem for the large kernel by simply not using chunking when the chunking factor is lower than the tile size. However, this is not an option for the loop-in-map kernel. Therefore, I wanted to explore how to improve the performance of transpose in the edge cases.

Improving Performance on Low Size in Either Dimension

I created a new transpose kernel to handle the cases where either dimension is smaller than the tile size. In the case of a low height of the input array, the new transpose kernel will read $m \times tilesize$ elements of each row instead of just $tilesize$, where $m = \lfloor \frac{tilesize}{height} \rfloor$. The implementation of this kernel is listed in appendix C.

The new transpose kernel should only be used in the cases where $m \geq 2$, which covers the edge cases from before. The performance improvement from using the new kernels to handle low width and low height, can be seen in Figure 5.9. We can see that it is has huge performance improvement compared to the normal 16x16 kernel for the edge cases, but that it is not as fast as the good cases for the normal 16x16 kernel.

An important detail for this new transpose is that it does not help in the case where the height falls with in $\frac{tilesize}{2} < height < tilesize$, because we cannot use $m \geq 2$. We know that the percentage of active threads in these cases must be strictly greater than 50%. In Figure 5.9 we can see that using 50% active threads, the $[2^3][2^{23}]$ entry, does give worse results but they are within a reasonable margin from the best performance. An other aspect of this problem is using a height of $tilesize + 1$, were we can effectively transpose the first $tilesize$ rows, but will inefficiently transpose the last row (because again we will use $tilesize \times tilesize$ threads to do so). Only using powers of two for the number of segments and segment size hides these details, which is why I wanted to bring attention to them.

We could improve things a bit by noting that transposing either a $[1][n]$ array or, a $[n][1]$ array, will not change the layout of the data in the array. We can simply skip the transpose altogether (if we need two separate arrays, we can use a memory copy; on the GTX 780 Ti this takes approximately $2000\,\mu s$ for this amount of data).

To the best of my knowledge, this approach to transposing an input array with low width or low height is novel.

5.3 Final Performance

We will now test the performance of the three kernels when computing a commutative and a non-commutative reduction. This means that we will include the cost of a transpose for the loop-in-map kernel, and when using the large kernel with chunking greater than one in the non-commutative case. We will use the 16x16 transpose kernel.

I have developed a simple algorithm to choose which kernel to use for a configuration of a number of segment and a segment size, which can be seen in Listing 5.4. The algorithm is based on two tuning parameters: number of threads for full hardware utilization, and the group size. It will prefer using the loop-in-map kernel if there are enough segments to fully utilize the hardware, otherwise it will prefer using the large kernel over the small kernel, if two segments do not fit within a group. We will also evaluate how good this algorithm is at choosing the optimal kernel for a configuration of number of segments and segment size.

When there is only one element per segment, the semantics of our reduction allows us to either use the input array as the output array, or to simply perform a memory copy. Because a redomap can apply a function to input elements before the reduction is performed, I have not used this optimization.

If the first reduction produces multiple intermediate results, I have not included the time it would take to perform the second reduction. I allow myself this freedom, because the cost is so low. In the worst case we will generate 240 intermediate results, and we can sum those up in $< 10\,\mu s$.

I have not included the time it would take to allocate and free such intermediate arrays, which is in the range of $150\,\mu s$ to $200\,\mu s$ on the GTX 780 Ti. I allow myself this freedom, because a smart memory management system could be able to optimize this away.

5.3.1 Commutative

Figure 5.10 shows the final performance of computing the commutative segmented sum, on 2^{20} and 2^{26} 32-bit floating points. We use the large kernel, the small kernel, and the loop-in-map kernel with transposition. When the chunking factor is 1, I assume the cost of a transpose is 0. The circles in Figure 5.10 mark the kernel that would be chosen by the algorithm from Listing 5.4. We can see that the choice is not always optimal:

```
segmented_reduction([number of segments][segment size] input,
                    [number of segments] output) {
    if (number of segments >= number of threads for full
        utilization) {
        input_t = transpose(input)
        loop-in-map_kernel(input_t, output)
    } else if (segment size > group size/2) {
        <...>
        if (non-commutative && chunking > 1) {
            input = transpose(input)
        }

        if (number of groups per segment > 1) {
            tmp = malloc([number of segments][number of groups per
                segment])
            large_kernel(input, tmp)
            segmented_reduction(tmp, output)
        } else {
            large_kernel(input, output)
        }
    } else {
        small_kernel(input, output)
    }
}
```

Listing 5.4: Algorithm to choose which kernel to use for a configuration of a number of segment and a segment size, based on two tuning parameters: number of threads for full hardware utilization, and the group size

---

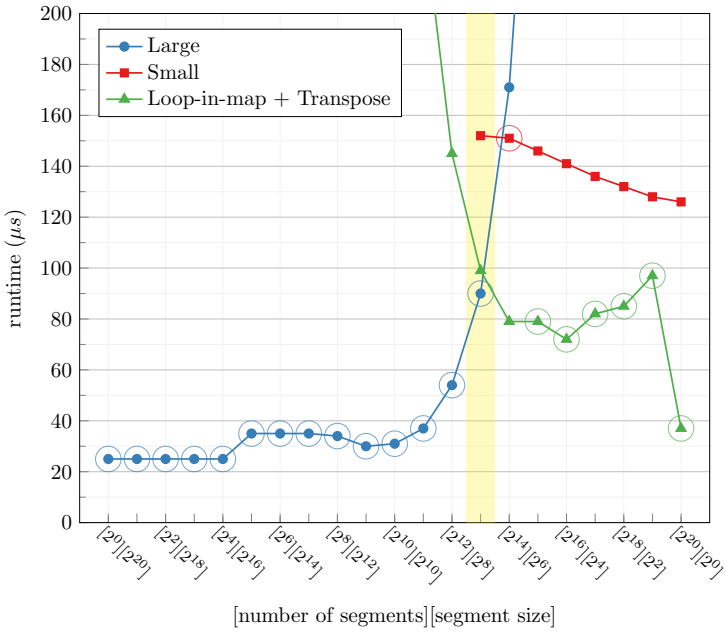
- Figure 5.10a shows that in the 2^{20} test case, we could get better performance by using the loop-in-map+transpose kernel for the $[2^{14}][2^{6}]$ configuration instead of the small kernel, otherwise it is on point.
- Figure 5.10b shows that in the 2^{26} test case, we could get better performance by using the large kernel a bit longer, instead of switching to the loop-in-map+transpose at $[2^{15}][2^{11}]$ it would be optimal to switch at $[2^{19}][2^{7}]$. The cost of the transpose is simply too high.

We can amend these sub-optimal choices by changing the tuning parameter for the number of threads for full hardware utilization.

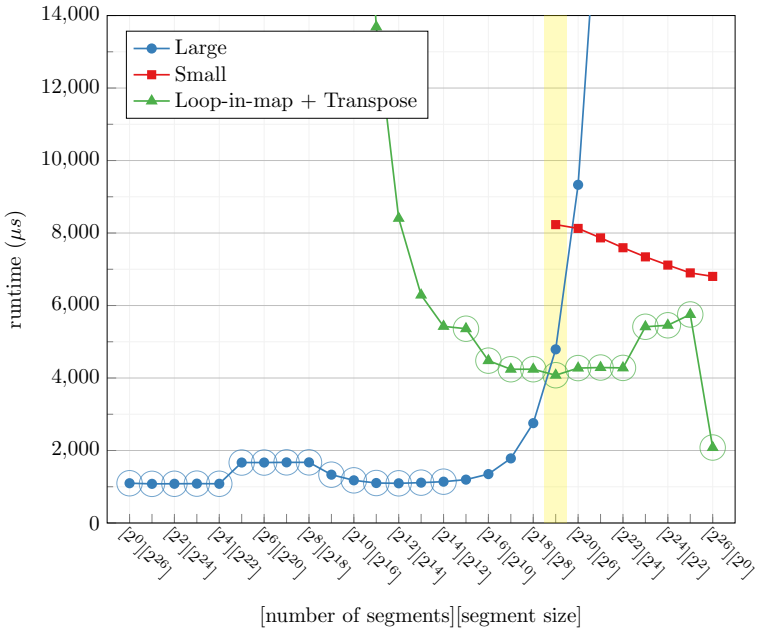
However, in my opinion the two cases are semantically different. Instead of tuning the number of threads for full hardware utilization as a "hack", we could also add a new rule specific to commutative reductions, that would prefer the large kernel over the loop-in-map kernel if the chunking factor was above a certain limit. This limit would then be yet another tuning parameter. This second approach would be able to correct the sub-optimal choices for the 2^{26} test case.

5.3.2 Non-commutative

Figure 5.11 shows the final performance of computing the non-commutative segmented sum, on 2^{20} and 2^{26} 32-bit floating points. We use the large kernel



(a) Using dataset of total 2^{20} 32-bit floating points.



(b) Using dataset of total 2^{26} 32-bit floating points.

Figure 5.10: Commutative performance. The circles mark the kernel that would be chosen by the algorithm from Listing 5.4.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size (chunking=1).

---

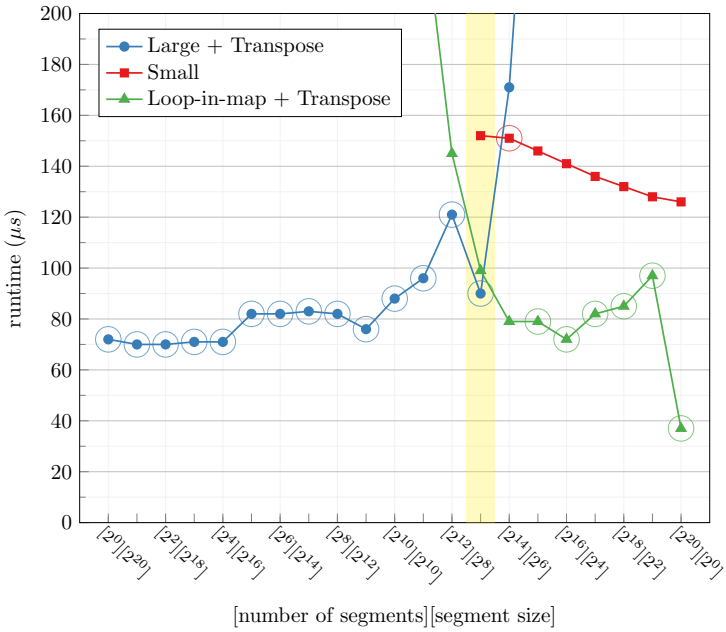
with transposition, the small kernel, and the loop-in-map kernel with transposition. When the chunking factor is 1, I assume the cost of a transpose is 0. The circles in Figure 5.11 mark the kernel that would be chosen by the algorithm from Listing 5.4.

We can see there are two case where it doesn't make the optimal choice:

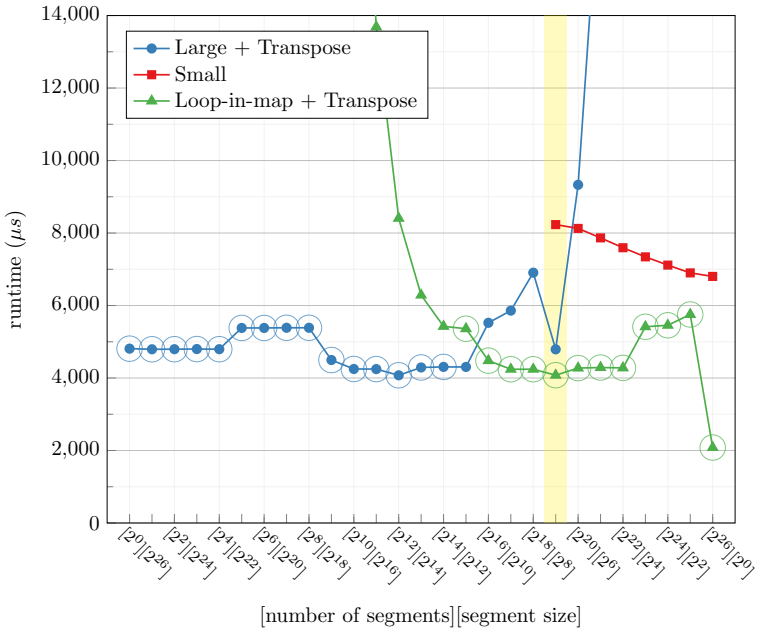
- Figure 5.11a shows that in the 2^{20} test case, we could get better performance by using transpose + loop-in-map, instead of the small kernel, for the $[2^{14}][2^{6}]$ configuration
- Figure 5.11b shows that in the 2^{26} test case, we could get better performance by using transpose + large, instead of transpose + loop-in-map, for the $[2^{15}][2^{11}]$ configuration.

As with the commutative reduction, these sub-optimal choices can be amended by changing the tuning parameter for the number of threads for full hardware utilization.

It is worth noting that for the 2^{26} test size, the cost of transposition when using less than 2^{8} segments is a bit high, because the chunking factor makes the



(a) Using dataset of total 2^{20} 32-bit floating points.



(b) Using dataset of total 2^{26} 32-bit floating points.

Figure 5.11: Non-commutative performance. The circles mark the kernel that would be chosen by the algorithm from Listing 5.4.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size (chunking=1).

16x16 transpose kernel hit the bad case shown in the middle of Figure 5.8[3]. If we used the 32x32 transpose kernel, we could save 1000 µs on the transpose in the configurations with less than 2^8 segments.

Chunking for Non-Commutative Reduction

In both Figure 5.11a and Figure 5.11b we can see that when using transpose + large kernel with chunking=2 (directly left of yellow bar), it is actually slower than when using chunking=1 (entry in yellow bar). For both test sizes this is also the case for chunking=4, and with the 2^{26} test size it is also true for chunking=8.

We could add a new rule specific for non-commutative reductions, that would prefer using the large kernel without chunking, if the chunking factor we could use was below a certain limit. This limit would then be yet another tuning parameter.

[3] I don't have any data for the cost of transposing an array of shape [30720][2185], so I used the values for $[2^{15}][2^{11}]$

5.4 Conclusion

In this chapter we have seen how using three different strategies for computing segmented reductions, allows us to get handle all configurations of number of segments and segment sizes. The *large* kernel is effective on large segments; the *loop-in-map* kernel is effective when are many segments, as it uses a single thread to process a segment; and the *small* kernel can be used in the remaining cases.

We have seen how reading multiple elements per thread significantly increases performance of the large kernel; as transposition is required for non-commutative reductions to use this technique, we have seen that commutative and non-commutative segmented reductions have different performance characteristics.

We have explored the cost of transposition, have seen that the standard transpose kernel has serious performance problems when handling segments with width or height smaller than the TILE size, and briefly seen a new transpose kernel that tries to mitigate this problem.

The simple algorithm to choose a kernel from Listing 5.4, will for any configuration of number of segments and segment size, select a reasonable kernel for both commutative and non-commutative reductions. We have seen that the optimal choice for the tuning parameters depends heavily on the problem size and whether the reduction is commutative or non-commutative:

- For the commutative reductions in Figure 5.10, setting the "optimal number of threads" to 2^{14} would be optimal for the 2^{20} case, but for the 2^{26} case setting it to 2^{19} would be optimal.
- For the non-commutative reductions in Figure 5.11, setting the "optimal number of threads" to 2^{14} would be optimal for the 2^{20} case, but for the 2^{26} case setting it to 2^{16} would be optimal.
- We have also seen that adding new rules and tuning parameters specifically for commutative and non-commutative reduction would allow us to address different classes of performance problems using their own tuning parameters, instead of using the optimal number of threads for everything as a hack.
- Computing a segmented redomap instead of a reduction will also influence the performance characteristics; for the segmented maximum segment sum problem (see section 3.3.2) the input array only uses one component per element, whereas the reduction operates over a 4-tuple. This changes the relationship between the time a transpose takes and the cost of performing different kinds of reductions.
- For the simple problem of segmented sum, we saw in subsection 5.2.3 that using a group size of 128 was beneficial, but it is unlikely that this will be the case for all problems and all input sizes.

I will not invest time in trying to statically finding the optimal values for these tuning parameters, but will leave this for future work on an auto-tuning

project. Therefore, and to keep things simple, I will keep my kernel-picking-algorithm as it is presented in Listing 5.4, even though it does not always make the optimal choice.

5.4.1 The Small Kernel

From the experiments we have investigated, it does not seem like the small kernel is very useful. However, we have not looked at the full picture: we have looked at performing the segmented reduction on arrays having more than a million elements in total. For example, consider an array with 1000 segments each having 64 elements; on a GPU that can handle 65536 active threads at one time, it is very likely that using the small kernel where the threads can all run at once, will be more efficient than using the loop-in-map kernel which can only use 1000 threads. We have also looked at the idealized performance, and completely ignored the cost of allocating and freeing a temporary arrays, which will be required by our transpose kernel, if we are not performing advanced memory management. On the GTX 780 Ti, I have observed the combined cost is within the range of $150\,\mu s$ to $200\,\mu s$. Even when using 2^{20} elements, if we allocate the temporary array and the result array in separate calls, this overhead would cause the small kernel to be much faster than the loop-in-map kernel.

5.4.2 Ideas for Future Work

After working on this prototype, I have considered the following additional aspects that could be explored to get even better performance:

- So far we have only looked at using a static group size, and segment sizes using powers of two. Using odd segment sizes might cause problems for some of the kernels. In the case of a non-commutative reduction, we have seen that the large kernel will only use approximate half its threads for a segment with $groupsize + 1$ elements. For the small kernel, if we use a segment size of $groupsize + 1$, we will at most be able to fit a single segment within a group, thereby wasting nearly half the threads.

  It could be interesting to see what benefits adjusting group size dynamically could give to these scenarios. However, doing this is a complicated endeavor: we need to have a detailed understand of the GPU device we are using, to know which group sizes can achieve high occupancy and which can't; we need to know the register and local memory usage of a kernel to determine which group are allowed; and we need to way to estimate the performance gains by for example increasing the group size of the small kernel a little, which will then lead to fewer groups being launched.

  An other example is using a high group size, such as 1024, which will be very inefficient for small segment sizes, such as 256. Dynamically adjust the group size could also help in this case.

- When using the large kernel, and it is reasonable to use multiple groups per segment, it might be beneficial to start by increasing chunking instead; the data on chunking in subsection 5.2.2 showed that even a chunking factor of 4 can increase performance significantly. When faced with a large segment, we could start by increasing the chunking factor until it reaches a limit, and only then start using multiple groups per segment. This limit would be yet an other tuning parameter.

  For example, reducing a single segment with 1024 elements is just as fast when using a group size of 128 and a chunking factor of 4, as using four groups of the same size: but by only using one group we don't need to use a temporary array or perform the recursive reduction.

Chapter 6

Implementation

In this chapter I will give the overview of how I implemented code generation for segmented redomaps and segmented reductions in the Futhark compiler. My implementation is heavily based on the prototype, described in the last chapter. I will not go into details with the code I added, but will try to give an intuition about how we arrive at the resulting Futhark code that is generated by my implementation.

The Futhark compiler is freely available at GitHub. I am describing the compiler as its state was at commit id `dd2d6651fd9ae9de6fcfe408ed02d54e4976b07e`.

My code generation for segmented redomaps is invoked by the function `regularSegmentedRedomap`, which is defined in the file `src/Futhark/Pass/ExtractKernels/Segmented.hs`. To enable my code generation for segmented redomaps we need to turn the global flag `newSegmentedRedomap` in the file `src/Futhark/Pass/ExtractKernels.hs`.

6.1 The Futhark Compiler

I will introduce a few details on the Futhark compiler, that needs to be established to understand how we can generate code for segmented redomaps.

First we will look at how redomaps are constructed by the fusion of maps and reductions; then we will look at the kernel extractor that turns an abstract syntax tree (AST) using second order array combinators (SOACs), such as maps and reductions, into an AST with kernels that can be run on the GPU.

The kernel extractor is only used in the pipeline for generating OpenCL code. There are other pipelines, such as the one for generating sequential C code – see [11] for more details.

The Futhark Compiler uses several other passes in these pipelines, but we will not go into details with these. For a more complete description, see [20].

```
fun foo (xs: [n]i32, a: i32, bs: [n]i32) : ([n]i32, i32, i32) =
  let ys = map (\x -> x+a) xs
  let sum_ys = reduceComm (+) 0 ys
  let zs = map (\(y, b) -> y*b) (zip ys bs)
  let prod_zs = reduceComm (*) 1 zs
  let xs' = map (\x -> x*2) xs
  in (xs', sum_ys, prod_zs)

fun segfoo (xss: [m][n]i32, a: i32, bs: [n]i32) :
                                    ([m][n]i32, [m]i32, [m]i32) =
  unzip (map (\xs -> foo(xs, a, bs)) xss)
```

Listing 6.1: Futhark's fusion can turn the expressions in the function `foo` into a single redomap.

6.1.1 Fusing Maps and Reductions

Here I will give an idea about how redomaps are constructed by fusing maps and reductions. If you want more details for how fusion for redomap works in Futhark see [18], and for details on fusion in Futhark in general see [20].

Listing 6.1 shows a contrived Futhark program. The expressions in the function `foo` will be fused into a single redomap. Recall from subsection 3.3.4 that a redomap has both a reduction function and a folding function. The folding function will be

```
\(acc_sum, acc_prod, x, b) ->  let y = x + a
                               let z = y * b
                               let new_acc_sum = acc_sum + y
                               let new_acc_prod = acc_prod * z
                               let x' = x * 2
                               in (x', new_acc_sum, new_acc_prod)
```

and the reduction function will be

```
\(acc_sum, acc_prod, y, z) ->  let new_acc_sum = acc_sum + y
                               let new_acc_prod = acc_prod * z
                               in (new_acc_sum, new_acc_prod)
```

with the neutral element (0,1).

The function `segfoo` will compute the segmented version of `foo`, so after fusion we will have redomap within a map – a segmented redomap.

Note that the `bs` array is only being mapped over by the inner map, so the value of `b` only depends on which iteration we are in of the inner map. We will say that the variable `b` is invariant in the outer map.

We will say that the computation of `xs'` is the *map-part* of the redomap. We will say that the computation of `ys` and `zs` is the *transform-part* of the redomap, as these are only used for the reduction. The map-part and the transform-part can overlap, although they don't in this example.

$\bar{q}^{(n)}$	::=	$q_1, \ldots, q_n$	(notation)
$\bar{q}$	::=	$q_1, \ldots, q_n$ for some n	(notation)
x	::=	`id`	(variable name)
z	::=	`id` | `Const`	(variable name or scalar value)
k	::=	`Const` | $[k_1, \ldots, k_n]$	(scalar or array value)
f	::=	`id`	(function name)
p	::=	x `:` τ	(typed variable)
t	::=	`i8` | `i16` | `i32` | `i64` | `bool` | `f32` | `f64`	(basic type)
τ	::=	t | $[z_1, \ldots, z_n]t$	(size-dep n-dim array type)
$l, \oplus$	::=	`(\(`$\bar{p}$`):(`$\bar{\tau}$`) ->` e`)`	(anonymous function)
e	::=	x	(variable)
	|	k	(scalar or array value)
	|	$(\bar{z}^{(n)})$	(n-tuple exp)
	|	x`[`z_1`, ..., `z_n`]`	(array indexing)
	|	$\odot\, z$	(unop)
	|	$z_1 \otimes z_2$	(binop)
	|	f `(`$\bar{z}$`)`	(function-call)
	|	`if` z `then` e_1 `else` e_2	(if)
	|	`let (`$\bar{p}$`) =` e_1 `in` e_2	(let-binding)
	|	x`[`z_1`,...,`z_n`] =` z	(in-place update)
	|	`loop ((`$\bar{p}^{(n)}$`) =` $(\bar{z}^{(n)})$`) =` `for` $z' < z''$ `do` e	(sequential do-loop)
	|	`iota` z	(the array $[0,\ldots,z-1]$)
	|	`reshape` $(\bar{z}^{(n)})$ x	(change shape of x to $(\bar{z}^{(n)})$)
	|	`replicate` $(\bar{z}^{(n)})$ x	(replicate the value x so outer-dim becomes $(\bar{z}^{(n)})$)
	|	`scratch` t $(\bar{z}^{(n)})$	(allocate memory for array with shape $(\bar{z}^{(n)})$ and type t)

Figure 6.1: Basic Expressions in Futhark, that are always allowed.

6.1.2 Kernel Extraction

The kernel extractor will take an AST using SOACs, and turn it into an AST only using the basic expressions that can be seen in Figure 6.1 and kernel launches. The kernel extractor looks for known patterns that can be turned into effective kernels. An example is turning `map f xs` into a kernel where each thread applies the function `f` to an element of the array `xs`.

Because Futhark is a functional language, the semantics of a kernel producing a result, is not to write an element to global memory as we have seen in the CUDA examples so far; Instead, kernels can return values using different return-constructs, such as one value per group, or one value per thread — we will cover these as we use them. When generating the final OpenCL code, these kernel return statements will indeed be turned into global memory writes, but it is not the semantics of the program at this point.

Kernels can use special expressions, an overview is given in Figure 6.2 and more details are given here:

`SplitSpace` o w i c is used when multiple threads should read multiple elements from an array, to calculate the number of elements thread i should read. The chunking factor c is the maximum number of elements

w	::=	`id` | `Const`	(width)
i	::=	`id`	(thread specific index)
o	::=	Strided z | Contiguous	(ordering)
e	::=	`SplitSpace` $o\ w\ i\ c$	(number of elements thread i should read, at most c — total elements is w.)
	|	`Combine` $i\ w\ t\ x$	(local memory array of type t and outer size w, with $arr[i] = x$ if $i < w$)
	|	`GroupReduce` $w\ \oplus\ (\bar{z}^{(n)})\ (\bar{x}^{(n)})$	(group-wide cooperative reduce)
	|	`GroupScan` $w\ \oplus\ (\bar{z}^{(n)})\ (\bar{x}^{(n)})$	(group-wide cooperative scan)
	|	`thread_id`	(the index of this thread, within its group)
	|	`group_id`	(the index of the group this thread belong to)

Figure 6.2: Special Futhark expressions that can only be used inside kernels.

each thread should read, out of the total number of elements w. The ordering o will either be Strided or Contiguous, and is used to handle how to distribute the last elements when the total number of elements w is not evenly divided by the chunking factor c.

`Combine` $i\ w\ t\ x$ will create a local memory array to hold w values of type t. All threads in a group must participate because synchronization barriers are used. If thread i has an index that is lower than w, the value x from the thread will be put at index i of the new local memory array.

`GroupReduce` $w\ \oplus\ (\bar{z}^{(n)})\ (\bar{x}^{(n)})$ performs a cooperative reduce within a group over the arrays $(\bar{x}^{(n)})$, with the reduction operator $\oplus$, and neutral element $(\bar{z}^{(n)})$. The arrays $(\bar{x}^{(n)})$ must have been brought into local memory by `Combine`. All threads in a group must participate because synchronization barriers are used.

`GroupScan` $w\ \oplus\ (\bar{z}^{(n)})\ (\bar{x}^{(n)})$ performs a cooperative scan within a group over the arrays $(\bar{x}^{(n)})$, with the scan operator $\oplus$, and neutral element $(\bar{z}^{(n)})$. The arrays $(\bar{x}^{(n)})$ must have been brought into local memory by `Combine`. All threads in a group must participate because synchronization barriers are used.

For the purpose of this thesis, we will not go into details with the kernel extractor. An important detail is that by default kernel extract will *only* use my function to handle segmented reductions by default.

There is a feature in the compiler currently under development called *versioned-code*, that at runtime can choose between multiple kernels performing the same computation. This feature is disabled by default. When enabled, for segmented reductions and segmented redomaps, the compiler will generate *both* a loop-in-map kernel and a segmented reduction kernel using either my function (if this has been enabled) or by using the segmented scan approach.

The function to generating code for segmented reductions and segmented redomaps should be able to handle an arbitrary number of outer maps, and be able to handle invariant variables in the folding function. Currently the

kernel extractor will not use our function to generate code, if any variables in the reduction operator of the redomap are invariant in any of the maps.

After the kernel extraction pass, a *kernel babysitter* pass is run. The kernel babysitter can detect that we are using chunking to read a contiguous range of elements of an array, and it will insert a transpose before the kernel, and apply a transformation to the indexing to match the new transposed structure. This will ensure memory coalesced access. This means that when our function generates code, we do not manually have to insert a transpose statement.

6.2 Implementation of Kernels

We will now study how I implemented the large and small kernels from the prototype (chapter 5). I will not implement the loop-in-map kernel, as this is covered by the versioned-code feature I introduced in the last section. Versioned-code is not enabled in the Futhark compiler by default, but we will superficially see the effect it has, when evaluating benchmarks in chapter 7.

The function I created gets a redomap as one of its input, but it is always possible to distinguish between a redomap and a reduce, by looking at the return type of the folding function. If there are as many values returned from the folding function as there are neutral elements, then we are in fact dealing with a reduction; if there are more, we are dealing with a redomap.

6.2.1 Large

Recall from subsection 5.1.2 that we can use large kernel to let multiple groups process a single segment, and that we can use chunking to let each thread process multiple elements.

```
map (\xs -> let ys = map f xs
            let zs = map g xs
            let red = reduce ⊕ ne zs
            in (red, ys)
    ) xss
```

I will give an overview of the Futhark code that is generated by my implementation to run the larger kernel. To keep things simple, we will consider the segmented redomap resulting from the code above.

Algorithm 6.1 shows pseudo-code as it would be generated by my implementation. We start by initializing variables, such as calculating the number of groups to use per segment, and the chunking factor to use. We also allocate global memory to store the results of the map-part (i.e., the results of applying f).

Then comes the kernel, that all threads will run:

- In "Part 1" of the kernel in Algorithm 6.1 we calculate the segment index (what segment are we working on), and this thread's index out of all

Algorithm 6.1 Large Kernel

Input: *Input array* inarr : [num_seg][seg_size] t_{in}
Functions: $f : t_{in} \to t_{mapout}$ $g : t_{in} \to t_{red}$ $\oplus : t_{red} \to t_{red} \to t_{red}$

num_groups_per_seg $\leftarrow \lceil \frac{\text{opt\_num\_groups}}{\text{num\_seg}} \rceil$
chunking $\leftarrow \lceil \frac{\text{seg\_size}}{\text{num\_groups\_per\_seg} \times \text{group\_size}} \rceil$
threads_for_seg $\leftarrow$ group_size $\times$ num_groups_per_seg
num_groups $\leftarrow$ num_seg $\times$ num_groups_per_seg
if commutative **then**
    stride $\leftarrow$ threads_for_seg
    ordering $\leftarrow$ Strided threads_for_seg
else
    stride $\leftarrow$ 1
    ordering $\leftarrow$ Continguous
end if
total_elems $\leftarrow$ num_seg $\times$ seg_size
mapout $\leftarrow$ `Scratch` (t_{mapout}) (total_elems) ▷ allocate global memory
function LARGEKERNEL:
    ▷ Part 1: Calculate indexes, offset to read from, and number of elements to read
    seg_index $\leftarrow$ group_id / num_groups_per_seg
    tid_within_seg $\leftarrow$ thread_id + group_size $\times$ (group_id % num_groups_per_seg)
    if commutative **then**
        offset $\leftarrow$ seg_index $\times$ seg_size + tid_within_seg
    else
        offset $\leftarrow$ seg_index $\times$ seg_size + tid_within_seg $\times$ chunking
    end if
    num_to_read $\leftarrow$ `SplitSpace` (ordering) (seg_size) (tid_within_seg) (chunking)
    ▷ Part 2: Create a "view" for the slice this thread should process (not materialized)
    input_slice $\leftarrow$ inarr[offset : num_to_read * stride]
    mapout_slice $\leftarrow$ mapout[offset : num_to_read * stride]
    ▷ Part 3: Apply the folding function
    acc $\leftarrow$ neutral_element
    for i $<$ num_to_read **do**
        x $\leftarrow$ input_slice[i]
        y $\leftarrow f(x)$
        z $\leftarrow g(x)$
        mapout_slice[i] $\leftarrow$ y ▷ inplace update of global memory
        acc $\leftarrow$ acc $\oplus$ z
    end for
    ▷ Part 4: Bring values into local memory, and perform coorperative reduce in group
    elemarr $\leftarrow$ `Combine` (thread_id) (group_size) (t_{red}) (acc)
    redresult $\leftarrow$ `GroupReduce` (group_size) ($\oplus$) (neutral_element) (elemarr)
    ▷ Part 5: Return results
    return *(only thread_id = 0)* redresult
    return mapout_slice *(at indexes)* [offset : num_to_read * stride]
end function
redres, mapres $\leftarrow$ <<<num_groups, group_size>>> LARGEKERNEL

Output: redres : [num_groups] t_{red} , mapres : [total_elems] t_{mapout} , num_groups_per_seg

the threads working on this segment. We use those values to calculate the offset for the first element this thread should read; this calculation depends on the whether the reduction is commutative. Lastly we use `SplitSpace` to compute the number of elements this thread should read.

- In "Part 2" of the kernel in Algorithm 6.1 we create an array slice of the input array, and for the output array that will hold the result of the map-part. Creating these array slices does not mean we store those elements in thread-local memory, it is simply a way to be able to index the global array easily in a loop: When accessing element i of the array, from a slice of [`offset` : `count` * `stride`], we need to get element `offset` $+ i \times$ `stride`.
- In "Part 3" of the kernel in Algorithm 6.1 we run the folding function over all elements of the input slice. We perform an in-place update of the `mapout_slice` in each iteration (and thus write to global memory). For simplicity, I expressed this as an imperative for loop here, but in Futhark we can express these lines as a `do loop` (as described in subsection 3.2.3).
- In "Part 4" of the kernel in Algorithm 6.1 we bring the reduction results from all threads into local memory, and perform the cooperative group-wide reduction.
- In "Part 5" of the kernel in Algorithm 6.1 we use the special Futhark constructs to return results from the kernel. Thread 0 of each group will return the result of the reduction, and all threads will return their `mapout_slice`.

In the last line of Algorithm 6.1 we launch the kernel, which will give us two arrays as the result. Assigning `mapres` might seem a bit odd at first; however, as we are still dealing with a functional program, it makes sense to assign the result to a new variable. It is the responsibility of another pass to see that we do in fact not need to copy the elements from `mapout` (the global array we allocated before the kernel) to `mapres`, but that we can simply use the same memory for the two arrays. Currently, this optimization is not performed, so each thread will copy all its elements from `mapout_slice` to the correct position in `mapres`[1].

Launching Too Many Groups

There is a serious limitation on the suitable values to use for the tuning parameters for group size (`group_size`) and for optimal number of groups of this size (`opt_num_groups`). When processing a single segment, regardless of the segment size, we will launch `opt_num_groups` groups to process it. If using a small segment size, this is clearly not optimal, as we will launch many more groups than is needed. These "extra" groups will not read any input elements, but they will still perform the group-wide reduction, thus taking up precious processing time.

The risk of this happening is only present when using a chunking factor of 1. In this case we can limit the number of groups launched to the number of

[1] unless gcc, which compiles the OpenCL code, performs an optimization

groups it will take to read all input elements. We can do this by inserting the following snippet between line 2 and 3 in Algorithm 6.1

---

if chunking $= 1$ **then**
  num_groups_per_seg $\leftarrow \min\left(\text{num\_groups\_per\_seg}\ ,\ \left\lceil \frac{\text{seg\_size}}{\text{group\_size}} \right\rceil\right)$
end if

Optimizing One Group Per Segment

When only using one group per segment, the calculations for `seg_index` and `tid_within_seg` from "Part 1" of Algorithm 6.1 could be simplified: we could replace them by the following code,

---

seg_index $\leftarrow$ group_id
tid_within_seg $\leftarrow$ thread_id + group_size

---

this removes one division, one multiplication, and one modulo operation, which would otherwise be performed by each thread.

My implementation will check if we are only launching one group per segment, and then use a specialized kernel exploiting this optimization. When using a high chunking factor, this optimization will not improve performance much, because most of the time is spend in the folding function loop. However, when each thread only processes one element, removing these three expressions should reduce the percentage of expressions executed by each thread significantly.

I have not measured how much this improves performance.

Optimizing Chunking=1 For Non-commutative Reductions

As explained above, the kernel babysitter pass will insert a transpose when it detects that we are accessing a contiguous slice of a global memory array. This is a static decision made before running the program, so even when using chunking=1, the transpose will still happen.

To overcome this we could create a special kernel, that is used when the chunking factor is 1: This kernel should only read a single element from the global input array, without creating a slice. In this way we will not perform the transpose.

I have currently not implemented this optimization.

6.2.2 Small

Recall from subsection 5.1.3 that the small kernel will be used to process multiple segments within one group. Each thread will read a single element from

the input array, apply the folding function, and then perform a cooperative segmented scan within the group. We will only assign whole segments to a group, so if a whole number of segments does not completely fill the group, some of the threads will have no work to do. We will call these threads *wasted*.

The wasted threads must participate in all `Combine` and `GroupScan` expressions, because of the synchronization barriers. We will calculate a variable to indicate if a thread is *active* or wasted. If the thread is not active, we will use the neutral element as the value to the `Combine` expression, as this will be a safe value to use in the `GroupScan`.

```
map (\xs -> let ys = map f xs
            let zs = map g xs
            let red = reduce ⊕ ne zs
            in (red, ys)
    ) xss
```

I will give an overview of the Futhark code that is generated by my implementation to run the small kernel. We will use the same underlying segmented redomap as we did when examining the large kernel: the segmented redomap resulting from the code above.

I will use $\oplus_{flag}$ to denote the reduction operator after it has been transformed to work as a segmented scan operator also taking in a flag value for each element, as described in subsection 3.4.1.

Algorithm 6.2 shows pseudo-code as it would be generated by my implementation. We start by initializing variables, such as calculating the number of whole segments that can fit within one group, and how many threads of a group will be active. We must take special care for the last group, because the number of segments it should process might be different from the other groups: for example, if we can process 3 segments per group, and there are 10 segments, we need 4 groups – but the last group will only process a single segment.

When returning values, only the active threads will return a value for the map-part, and the last thread in a segment will return the reduction value for its segment. There is no kernel-return-constructs in Futhark that maps directly to this, so instead we will use a kernel-return-construct that lets us write a value from each thread to a specific location in already allocated global memory. For all threads that should not return a value, we will use an invalid offset (-1), which causes the write to never happen. However, due to the functional nature of Futhark, we have to provide both a value and an offset for all threads.

Then comes the kernel, that all threads will run:

- In "Part 0" of the kernel in Algorithm 6.2, we calculate the number of active threads for this group, and calculate if this thread should be active or not.
- If the thread is active, we will perform the following steps:
  - In "Part 1" of the kernel in Algorithm 6.2, we calculate the segment index (what segment are we working on), and this thread's index out of all the threads working on this segment. We use those values to calculate the offset for the element this thread should read.

Algorithm 6.2 Small Kernel

Input: *Input array* inarr : [num_seg][seg_size] t_{in}
Functions: $f : t_{in} \to t_{mapout}$ $g : t_{in} \to t_{red}$ $\oplus_{flag} : (\mathsf{bool}, t_{red}) \to (\mathsf{bool}, t_{red}) \to (\mathsf{bool}, t_{red})$

num_seg_per_group $\leftarrow \left\lfloor \frac{\text{group\_size}}{\text{seg\_size}} \right\rfloor$
active_threads_per_group $\leftarrow$ num_seg_per_group $\times$ seg_size
if num_segments $\%$ num_seg_per_group $= 0$ **then**
    seg_in_last_group $\leftarrow$ num_seg_per_group
else
    seg_in_last_group $\leftarrow$ num_segments $\%$ num_seg_per_group
end if
active_threads_in_last_group $\leftarrow$ seg_in_last_group $\times$ seg_size
num_groups $\leftarrow \left\lceil \frac{\text{num\_seg}}{\text{num\_seg\_per\_group}} \right\rceil$
total_elems $\leftarrow$ num_seg $\times$ seg_size
redout $\leftarrow$ `Scratch` (t_{red}) (num_seg) ▷ allocate global memory
mapout $\leftarrow$ `Scratch` (t_{mapout}) (total_elems) ▷ allocate global memory

function SMALLKERNEL:
    ▷ Part 0: Calculate number of active threads in this group
    if group_id $=$ num_groups -1 **then**
        active_threads_this_group $\leftarrow$ active_threads_in_last_group
    else
        active_threads_this_group $\leftarrow$ active_threads_per_group
    end if
    isactive $\leftarrow$ thread_id $<$ active_threads_this_group

    if isactive **then**
        ▷ Part 1: Calculate indexes, and offset to read element from
        seg_index $\leftarrow$ ($\left\lfloor \frac{\text{thread\_id}}{\text{seg\_size}} \right\rfloor$) + (group_id $\times$ num_seg_per_group)
        tid_within_seg $\leftarrow$ thread_id $\%$ seg_size
        offset $\leftarrow$ seg_index $\times$ seg_size + tid_within_seg

        ▷ Part 2: Apply the folding function
        acc $\leftarrow$ neutral_element
        x $\leftarrow$ inarr[offset]
        y $\leftarrow f(x)$
        z $\leftarrow g(x)$
        acc $\leftarrow$ acc $\oplus$ z
        mapres $\leftarrow$ y
        mapoffset $\leftarrow$ offset
    else
        acc $\leftarrow$ neutral_element
        mapres $\leftarrow$ dummy_value
        mapoffset $\leftarrow$ -1
    end if

    ▷ Part 3: Bring values into local memory, and perform coorperative segmented scan in group
    elemarr $\leftarrow$ `Combine` (thread_id) (group_size) (t_{red}) (acc)
    isfirst_in_seg $\leftarrow$ (thread_id $\%$ seg_size) $= 0$
    flagarr $\leftarrow$ `Combine` (thread_id) (group_size) (bool) (isfirst_in_seg)
    scanresult $\leftarrow$ `GroupScan` (group_size) ($\oplus_{flag}$) (`false`, neutral_element) (flagarr, elemarr)

    ▷ Part 4: Calculate which threads writes reduction result
    islast_in_seg $\leftarrow$ (thread_id $\%$ seg_size) $=$ (seg_size -1)
    if isactive **and** islast_in_seg **then**
        redres $\leftarrow$ scanresult[thread_id]
        redoffset $\leftarrow$ seg_index
    else
        redres $\leftarrow$ neutral_element
        redoffset $\leftarrow$ -1
    end if

    ▷ Part 5: Return results
    write redres *(to)* redout *(at index)* redoffset
    write mapres *(to)* mapout *(at index)* mapoffset
end function

redres, mapoutres $\leftarrow$ <<<num_groups, group_size>>> SMALLKERNEL

Output: redres : [num_groups] t_{red} , mapoutres : [total_elems] t_{mapout}

  - In "Part 2" of the kernel in Algorithm 6.2, we run the folding function on the input element. Then we assign the result of the map-part to the `mapres` variable that will be used in the end of the kernel to write to memory. We set the `mapoffset` offset-variable for writing the map-part result to the same offset as we read the input element from.

- If the thread is not active, we provide a value for the accumulator `acc` that will be used for the cooperative scan, which all threads must participate in. We also provide a dummy value for the map-part result `mapres`, and set the offset `mapoffset` to negative one, so the dummy value is not written to memory in the end of the kernel.

- In "Part 3" of the kernel in Algorithm 6.2, we bring the reduction results from all threads into local memory. We calculate the flag value for each thread, and bring those into local memory. Then we can perform the cooperative group-wide scan, using the modified reduction operator.

- In "Part 4" of the kernel in Algorithm 6.2, we assign the reduction values that should be written at the end of the kernel. For the last thread in each segment, we will set the reduction result variable `redres` to the correct value from the `GroupScan` result array. We will also set the offset `redoffset` to the segment index.

  For all other threads we provide a dummy value for the reduction result variable (the neutral element), and set the offset to negative one, so the dummy value is not written to memory in the end of the kernel.

- In "Part 5" of the kernel in Algorithm 6.2, we return the results, by writing the reduction result variable to memory if the reduction offset is within bounds, and by writing the map-part result variable to memory if the map offset if within bounds.

In the last lines of Algorithm 6.2, we simply launch the kernel. Note that we are getting two result arrays from the computation. As in the case for map-part result from the large kernel, we already have a variable for the global memory that has been updated by the kernel, but we still assign the result of the kernel to two new variables. However, as we are using the write kernel-return-construct, we do *not* perform a copy between the two arrays, but simply alias them.

6.2.3 Handling Tuple-of-Arrays

We have not covered how to handle the cases where either the input, the map-part, or the reduction operates on tuples; as we saw in the example from Listing 6.1.

Recall from subsection 3.2.2 that we transform arrays-of-tuples into tuples-of-arrays. Therefore, to handle a tuple, instead of reading one element from one input array, we would simply read multiple elements from multiple input arrays. The same goes for writing the elements for the map-part. When the

a index	b index	c index	segment index
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
0	2	0	4
0	2	1	5
1	0	0	6
1	0	1	7
1	1	0	8
1	1	1	9
1	2	0	10
1	2	1	11

Table 6.1: Resulting indexes when using three outer maps, of size $a = 2, b = 3, b = 2$

reduction operates uses tuples, we will need to bring an array into local memory for each tuple-component.

If using tuples that contain many components, we might run into problems because each thread in a kernel uses too many registers, or because each thread needs to use a lot of local memory for all the components of the reduction.

6.2.4 Handling Invariant Variables

We have not covered how to handle variables that invariant in the outer maps, as `b` was in the example from Listing 6.1. The kernel extractor will pass the sizes of the outer maps to our function; it will also give us a list of variables that are invariant in some maps, what array we can get their value from, and which map to use for the index to this array.

```
map            -- over array of size 'a'
  map          -- over array of size 'b'
    map        -- over array of size 'c'
      redomap
```

Thus, we can handle invariant variables if we can compute the index of each of the outer maps, by using the segment index. As an example, consider a redomap that has three outer maps, as in the example above. If the sizes of the outer maps are $a = 2, b = 3, c = 2$, we should calculate indexes for each maps as shown in Table 6.1.

Algorithm 6.3 How we compute the final result of a segmented redomap, using the large kernel and small kernel

Input: *Input array* inarr : [num_seg][seg_size] t_{in}
Tuning Parameters: opt_num_groups , group_size

```
1: total_elems ← num_seg × seg_size
2: ▷ Compute first step of redoamp
3: if seg_size > group_size/2 then
4:     redres, mapres, new_seg_size ← LARGEREDOMAP(inarr)
5: else
6:     redres, mapres, ← SMALLREDOMAP(inarr)
7:     new_seg_size ← 1
8: end if
9: ▷ Recursively reduce the intermediate results, until one result per segment
10: while new_seg_size > 1 do
11:     tmp_inarr ← redres
12:     if new_seg_size > group_size/2 then
13:         redres , new_seg_size ← LARGEREDUCEONLY(tmp_inarr)
14:     else
15:         redres ← SMALLREDUCEONLY(tmp_inarr)
16:         new_seg_size ← 1
17:     end if
18: end while
```

Output: redres : [num_seg] t_{red} , mapres : [total_elems] t_{mapout}

We can compute these indexes using the following equations, using the segment index si:

$$\text{c index} = si \mod c$$
$$\text{b index} = \frac{si}{c} \mod b$$
$$\text{a index} = \frac{si}{c \times b} \mod a$$

we can generalize this formula to any number of outer maps.

6.2.5 Combining Kernels to Compute Final Result

As in the prototype, we will use multiple steps to compute the final result of a segmented redomap; if the kernel in the first step computes multiple reduction results for a segment, we will need to reduce those recursively. However, we cannot use the same kernel in both steps: For a redomap, the first kernel should compute both the map-part and the transform-part of the redomap, but we should not do this when recursively reducing the intermediate results.

Algorithm 6.3 shows pseudo-code for how we compute the final result of a segmented redomap, using the large kernel and small kernel. As in the prototype (section 5.3) we will prefer using the large kernel, and will only use the small kernel when we can fit at least two segments within one group. When using the small kernel, we will always compute the final result of the reduction part, so the variable `new_seg_size` will always be set to 1.

The while loop of Algorithm 6.3 can be implemented in Futhark using a *do-while-loop*. I did not document it in chapter 3, but it is like a do-loop: instead of having a predefined number of iterations, it continues until an expression is no longer true.

The result for the map-part returned by either the large or the small kernel will be a flat array, so we need to reshape it to become the correct shape (the shape of all the outer maps, and the segment size).

For simplicity, I use the same approach for both redomaps and reductions, although we could reuse the kernels when dealing with a reduction.

More Tuning Parameters

In the conclusion of the prototype chapter (section 5.4) we discussed how new tuning parameters could be used to optimize the performance of segmented reductions. You might be wondering why I did not implement these; but this is solely because we do not have any infrastructure to support autotuning yet.

As we discussed in the prototype chapter, the optimal values for all the tuning parameters will depend on the GPU device we use, the problem we are solving, and the problem size. So an autotuning solution is really the way forward for setting this tuning parameters.

Only Two Steps

In the current implementation, I made a significant simplification to Algorithm 6.3. The LARGEREDUCEONLY will always use a single group to process a segment. This ensures that we will always be able to complete the whole reduction using only two steps[2].

When the tuning parameter for optimal number of groups (of the specific group size) is fairly low, this is not a big problem. For example, for the GTX 780 Ti, if we use the 240 groups that will ensure full utilization, using a single group will be beneficial over using two groups and an extra step.

However, if using an enormous amount of groups as the optimal number of groups, this approach will clearly have its downsides. When dealing with a non-commutative reduction, this approach will require a transpose when the chunking factor used is greater than one; this can be more expensive than applying a two-step non-chunked reduction.

I envision that this problem can be better handled in the future, when we are more careful about choosing the chunking factor and the number of groups to use per segment.

[2] this also means that in the generated code, we will not see a do-while-loop, but instead an if expression

```
fun segsum (xss : [m][n]f32): [m]f32 =
  map (\xs -> reduceComm (+) 0.0f32 xs) xss

fun main (xsss : [l][m][n]f32): [l][m]f32 =
  map segsum xsss
```

Listing 6.2: A reduction over the inner array of a 3D array can be handled by my implementation.

```
fun add_if_smaller (const : i32) (acc : i32) (x : i32) : i32 =
  if x < const
  then acc + x
  else acc

fun main (xss : [m][n]i32, consts : [m]i32): [m]i32 =
  map (\c xs -> reduce (add_if_smaller c) 0 xs) consts xss
```

Listing 6.3: Invariant variables used in the reduction function of a redomap could be handled by my implementation, but the kernel extractor will conservatervely not use it.

6.3 Cases that Can and Cannot be Handled by My Implementation

In this section I will give an overview of which cases of segmented reduction can and cannot be handled by my implementation.

6.3.1 Cases That Can be Handled

My implementation can handle reductions over the inner array of all multidimensional arrays, no matter how many dimensions it has. For example, my implementation can handle the reduction over the inner array of a 3D array, shown in Listing 6.2.

A variable that is only defined in *some* of the maps in the reduction of a multidimensional array must be handled with care, as explained in subsection 6.2.4. If the invariant variable is used in the folding function of a redomap, my implementation will handle this properly; Listing 6.1 from the beginning of this chapter is an example of this.

6.3.2 Cases That Cannot be handled

If an invariant variable is used in the reduction function of a redomap, my implementation will not be used. My implementation would be capable of handling these, but currently the kernel extractor is conservative, and will not use my code generation function; I believe this is caused by the standard

```
fun main (xss : [m][n]i32, ys : [l]i32): ([m]i32, [m][n][l]i32) =
  unzip (map( \(xs : [n]i32) : (i32, [n][l]i32) ->
          let zs = map (\x -> map (\y -> x+y) ys) xs
          in (reduce (+) 0 xs, zs)
      ) xss)
```

Listing 6.4: A folding function in a redoamp that returns an array can currently not be handled by my implementation. In this example, `map (\y -> x+y) ys` will have to be computed for each element `x`

```
-- The reduction operator works on lists
fun vec_add (xs : [k]i32) (ys : [k]i32) : [k]i32 =
  map (\x y -> x + y) xs ys

fun main (xsss : [l][m][n]i32): [l][n]i32 =
  let zeros = replicate n 0 in
  map (\(xss : [m][n]i32) : [n]i32  -> reduce vec_add zeros xss)
      xsss
```

Listing 6.5: An example of a segmented reduction where the operand to the reduction operator are lists. Futhark will not use my implementation to generate code for such segmented reductions.

implementation having difficulty dealing with this case. An example of an invariant variable being used in the reduction function of a redomap, can be seen in Listing 6.3, where we compute the sum of all elements in an array `xs` that are larger than some constant `c` for pairs of constants and arrays.

My implementation cannot handle a folding function of a redomap that returns a list. This is only due to the fact that in the segmented scan for the small kernel, we must provide a dummy element of the correct type. Currently I have only implemented providing dummy values for primitive types. I have not encountered any problems on real programs due to this; however, I have a contrived example where this becomes an issue, shown in Listing 6.4. The function in Listing 6.4 will produce a folding function that will compute `map (\y -> x+y) ys` for the input element `x`, and therein lies the problem.

I will not claim that my implementation can handle reduction operators that takes a list as its input. For all such examples I have tried to create, none of them have caused my code generation to be used. One of the examples I have created is shown in Listing 6.5, where we add the inner arrays of a 2D array together, for all the 2D arrays contained in a 3D array.

6.4 Transpose

I implemented the new transpose kernel to handle input arrays with low width or low height in Futhark, that was mentioned earlier in subsection 5.2.4. This

greatly increased performance when using a low chunking factor. More details for the general idea behind this kernel can be found in appendix C.

To handle an input array that have width$= 1$ or height$= 1$, I perform a memory copy. Currently it is not possible to alias the memory, so we will not get a transpose cost of 0 as I assumed in the prototype, but a memory copy is still significantly faster than performing a transpose.

6.4.1 Padding

When using transpose for chunking, if the number of elements in the array is not a multiple of the number of threads, we will need to pad the array to transpose it as described in section 4.4. This makes the total cost of a transpose even bigger, as we first need to perform a memory copy of the entire array, and then perform the transpose. For simplicity, this padding-memory-copy is always performed, even when it is not needed.

The transpose kernel in Futhark has a parameter for max valid index (in both input and output), that could be used to avoid making this padded array, by only transposing the number of elements we actually need. However, if we avoid making the padded array, but the type information says that the array has the shape $[chunking][threads]$, then we need to propagate the fact that not all elements are addressable. Therefore, this feature is not used currently, but in the future the cost of a transpose could be reduced by enabling this.

6.5 Simple Performance Experiments

Now we will look at some simple performance experiments for my implementation. First off we will look at the performance of computing a commutative and a non-commutative segmented sum, and then we will look at the segmented maximum segment sum problem.

I will use the Futhark's default number of groups and group size, that is 128 groups with size 256. It is possible to specify these as arguments to OpenCL programs, by using `--group-size` and `--num-groups`; I will not try to find the optimal values for these, as discussed earlier.

As in the prototype, we will look at using a fixed total number of elements, and varying the number of segments and the segment size. I will keep using 2^{20} and 2^{26} as the total number of elements. The choice for using 2^{26} has historic reasons: When I started working on this thesis, Futhark could only read data from files in a textual format. This meant that for testing all powers of two as the number of segments for 2^{26} elements, we would need to generate 27 data files; each would take nearly 1 GiB of disk space, and reading all this data would be very slow. I created a binary input format for Futhark, and made the C runtime system able to read it to solve this problem, but stuck with the 2^{26} total elements.

```
fun main (xs: [n]f32): f32 =
  reduceComm (+) 0.0f32 xs
```

Listing 6.6: Computing a commutative segmented sum

```
fun main (xs: [n]f32): f32 =
  reduce (-) 0.0f32 xs
```

Listing 6.7: Computing a non-commutative "segmented sum". We use substraction to get the same operator intensity, as the Futhark compiler knows that addition is commutative.

6.5.1 Segmented Sum

In this section we will study the performance my implementation achieves for commutative and non-commutative reductions, by computing a segmented sum. The Futhark source program for the commutative segmented sum is shown in Listing 6.6.

As sum *is* commutative, and the Futhark compiler knows this, computing a non-commutative sum is not possible; therefore, we use subtraction for the reduction operator, even though this will not be any sensible computation, this means the two reduction operators have the same computational complexity. The implementation of the non-commutative reduction can be seen in Listing 6.7.

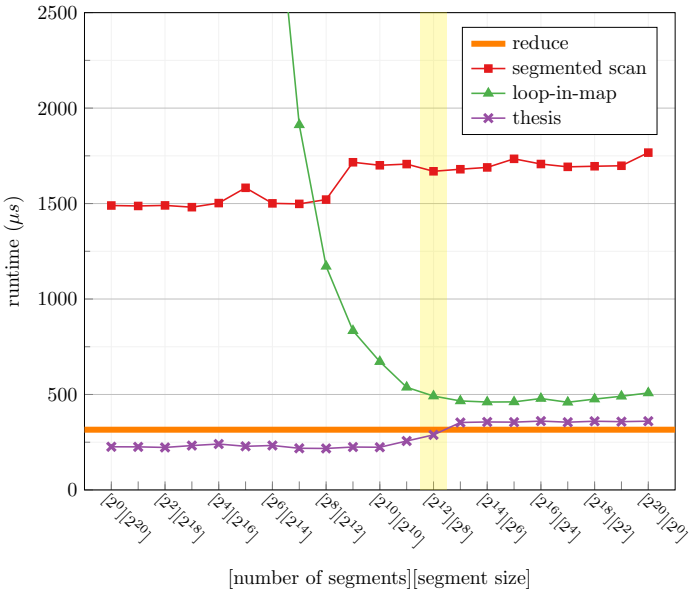
As the last piece of the puzzle, we will use the manual implementation for the loop-in-map strategy from Listing 5.1. Manually implementing it in the source language allows us to evaluate the performance of this strategy.

Commutative

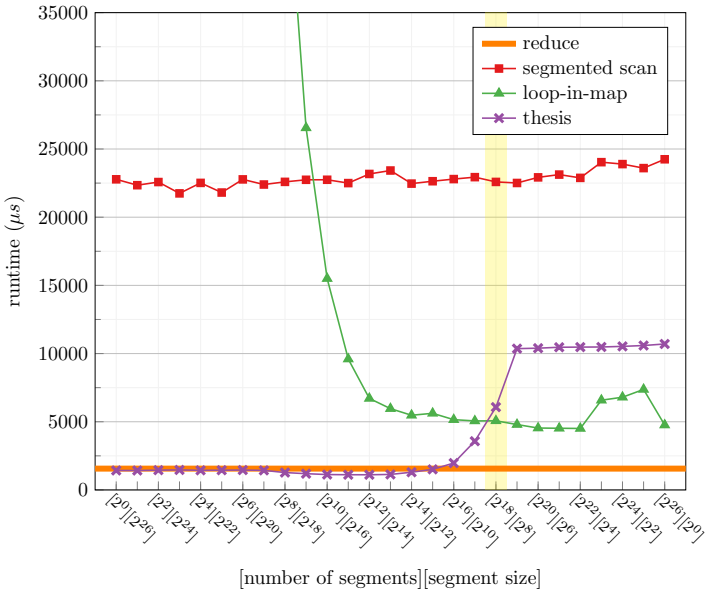
Figure 6.3 shows the performance of computing the commutative segmented sum, using 2^{20} and 2^{26} 32-bit floating points, on the GTX 780 Ti. The horizontal *reduce* line marks the performance of a one-dimensional reduction, *segmented scan* shows the performance when not using my implementation, *loop-in-map* is

```
fun main (xss: [m][n]f32) : [m]f32 =
  map (\xs ->
        loop (sum = 0.0f32) = for i < n do
            sum + xs[i]
        in sum
      ) xss
```

Listing 6.8: Computing a segmented sum by the loop-in-map strategy manually.



(a) Using dataset of total 2^{20} 32-bit floating points.



(b) Using dataset of total 2^{26} 32-bit floating points.

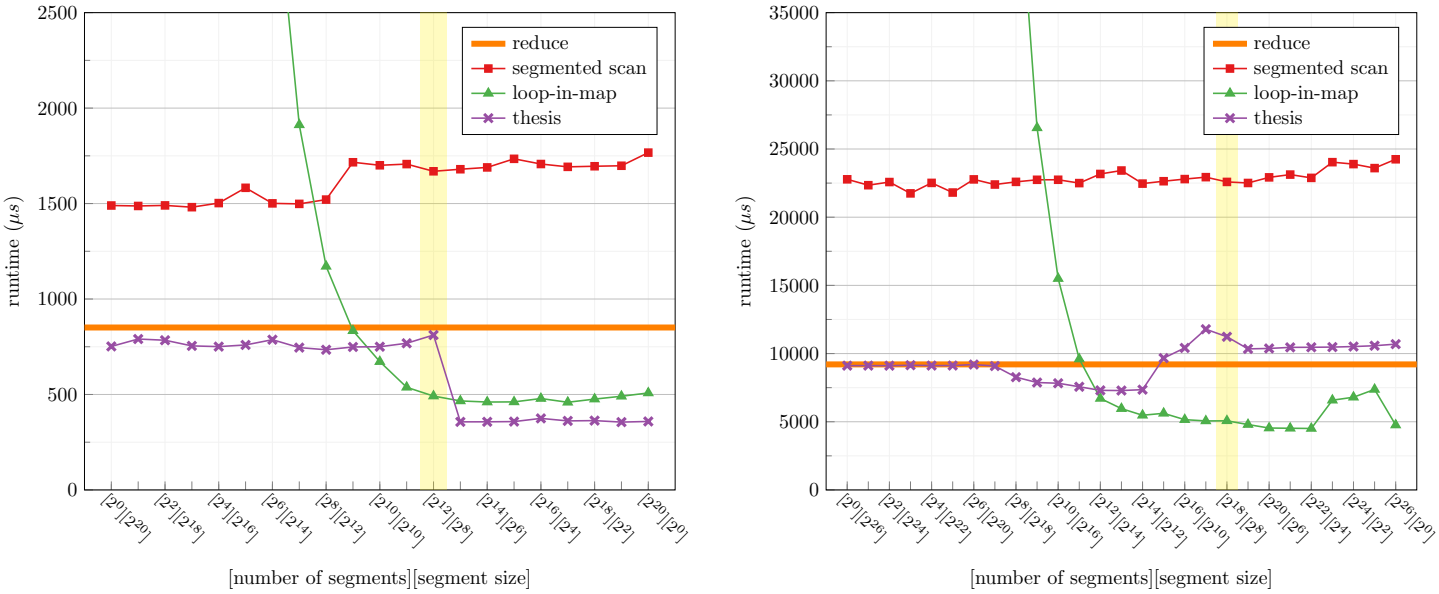
Figure 6.3: Performance for commutative segmented sum. The horizontal *reduce* line marks the performance of a one-dimensional reduction, *segmented scan* shows the performance when not using my implementation, *loop-in-map* is the performance for the loop-in-map strategy, and *thesis* shows the performance of using my implementation to generate code.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size for the larger kernel (chunking=1).

---

the performance for the loop-in-map strategy, and *thesis* shows the performance of using my implementation to generate code.

Figure 6.3a shows that for the 2^{20} case, my implementation outperforms the standard segmented scan implementation. We can also see that for all configurations of number of segments and segment size, my implementation is better or very close to the performance of the 1D reduction. The small kernel (used to right of the yellow bar) is so fast compared to the loop-in-map strategy, that we would not benefit from using the loop-in-map strategy for this case.

Figure 6.3b shows that for the 2^{26} case, my implementation outperforms the standard segmented scan implementation. We can also see that until the chunking factor drops to 4 (at 2^{10} elements per segment, two steps to the left of the yellow bar), we have performance comparable to a 1D reduction. In this case, we could gain a great runtime improvement by using the loop-in-map kernel, instead of just relying on the small kernel. The transpose required by the loop-in-map will have a slightly higher cost when there are less than 16 elements per segment, even though the new transpose kernel has greatly improved the performance we would otherwise have had; when there is only one element per segment, we can use a memory copy which is why it has a lower runtime.



(a) Using dataset of total 2^{20} 32-bit floating points.

(b) Using dataset of total 2^{26} 32-bit floating points.

Figure 6.4: Performance for non-commutative segmented sum. The horizontal *reduce* line marks the performance of a one-dimensional reduction, *segmented scan* shows the performance when not using my implementation, *loop-in-map* is the performance for the loop-in-map strategy, and *thesis* shows the performance when using my implementation to generate code.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size for the larger kernel (chunking=1).

Compared with the performance from the commutative reduction in the prototype, it can initially seem like there is extra overhead when using Futhark, but in the prototype I did not include the time for allocating and freeing the array for the intermediate values.

Non-commutative

Figure 6.4 shows the performance of computing the non-commutative segmented sum, using 2^{20} and 2^{26} 32-bit floating points, on the GTX 780 Ti. The horizontal *reduce* line marks the performance of a one-dimensional reduction, *segmented scan* shows the performance when not using my implementation, *loop-in-map* is the performance for the loop-in-map strategy, and *thesis* shows the performance when using my implementation to generate code.

In both cases we can see that my implementation always outperforms the standard segmented scan implementation. We can also see that the chunking factor does not have as large an impact for the performance of the large kernel as it did in the commutative case.

Figure 6.4a shows that for the 2^{20} case, my implementation always has

comparable performance with the 1D non-commutative reduction. As in the commutative case, the small kernel outperforms the loop-in-map strategy. However, there are a few configurations where using the loop-in-map strategy would yield better performance, specifically when there are more than 2^{10} segments and we still cannot use the small kernel.

Figure 6.4b shows that for the 2^{26} case, my implementation has a comparable performance with the 1D non-commutative reduction, although there are some deviations when the chunking factor drops (both because of the added cost of transpose, and the diminishing returns when using a lower chunking factor). In this case, we could gain a larger runtime improvement from using the loop-in-map kernel, starting from 2^{12} segments.

The fact that the large kernel is able to get bit better performance than the one dimensional reduction, is not very significant. Recall from the discussion on what chunking factor to use in the prototype chapter (subsection 5.2.2), that the number of groups used can impact the runtime of significantly. And sure enough, if we use `-group-size 1024 -num-groups 1024` we are able to get a runtime of 7626 µs for the 2^{26} case. This is not to say that this is the optimal launch configuration, just that it is perfectly possible to get better performance with the one-dimensional reduction as well.

An interesting observation is that the performance for the large kernel with chunking=1 in the commutative cases, is much better than the performance for the large kernel in any of the configurations for the non-commutative cases. By using the optimization of not using chunking at all in the non-commutative cases described in section 6.2.1, we can avoid the cost of transposition altogether, and might be able to improve performance of non-commutative reductions significantly.

6.5.2 Maximum Segment Sum

In section 3.3.2 we saw how we could use solve the maximum segment sum (MSS) problem using a non-commutative reduction with the `mss` function shown in Listing 6.9; the map and reduce will be fused into a redomap. We can compute a segmented MSS, by simply wrapping this content of `mss` in a map, as I have done in the `segmss` function.

I created a special version of the compiler that would use my implementation to generate code for segmented redomaps, without enabling versioned-code. This allows us to study the performance of my implementation for all configurations.

Figure 6.5 shows the performance of computing the segmented MSS, using 2^{20} and 2^{26} 32-bit integers, on the GTX 780 Ti. The horizontal *redomap* line marks the performance of the one-dimensional MSS; *loop-in-map* is the performance for the loop-in-map strategy, which is the default for segmented redomaps; and *thesis* shows the performance when using my implementation to generate code.

The loop-in-map code generated by the Futhark compiler will only need to write a single element to global memory for each segment. However, my implementation needs to write all four reduction results to memory. This can

```
fun max(x: i32) (y: i32): i32 =
  if x > y then x else y

-- (best, left, right, total)
fun redOp((bx, lx, rx, tx): (i32,i32,i32,i32))
         ((by, ly, ry, ty): (i32,i32,i32,i32)): (i32,i32,i32,i32) =
  ( max bx (max by (rx + ly))
  , max lx (tx+ly)
  , max ry (rx+ty)
  , tsx + tsy)

fun mapOp (x: i32): (i32,i32,i32,i32) =
  (max x 0, max x 0, max x 0, x)

fun mss(xs: []i32): i32 =
  let (x, _, _, _) = reduce redOp (0,0,0,0) (map mapOp xs)
  in x

fun segmss(xss: [m][n]i32): [m]i32 =
  map (\xs -> let (x, _, _, _) = reduce redOp (0,0,0,0) (map mapOp
      xs) in x) xss
```

Listing 6.9: Computing maximum segment sum, in both a one-dimensional and a segment version. The reduce and map will be fused into a redomap.

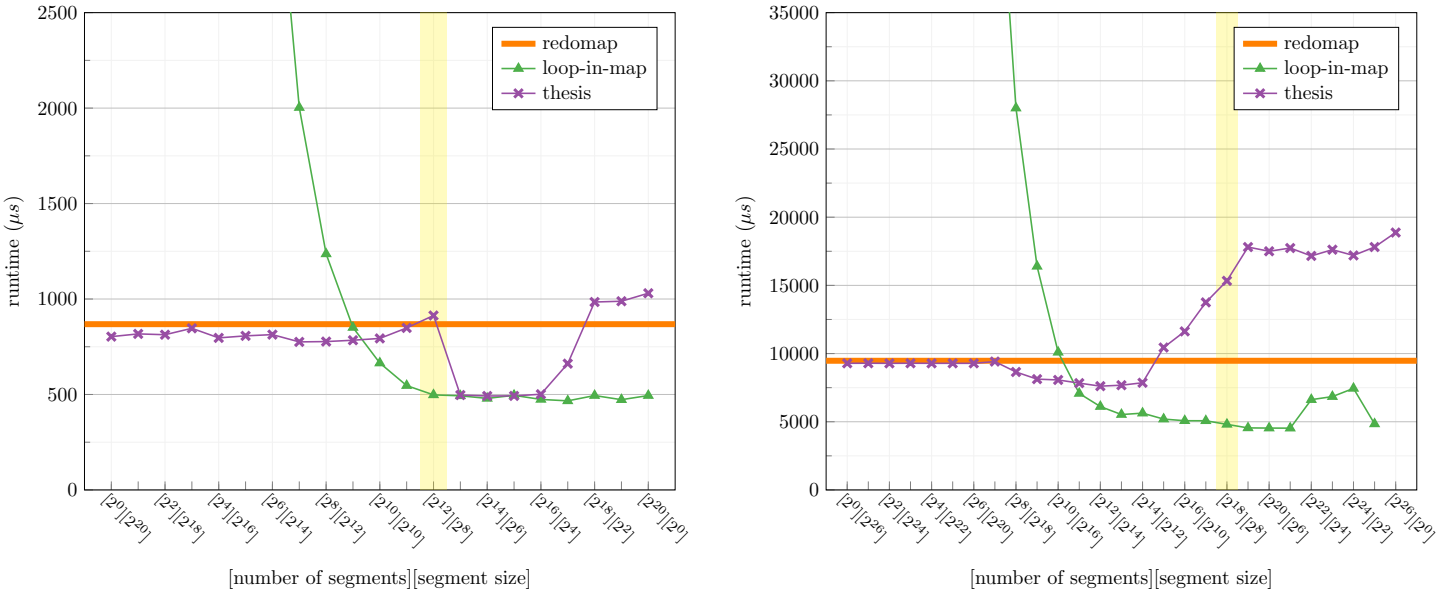
---

explain the seemingly strange behavior of the small kernel in Figure 6.5a: for the first four configurations to the right of the yellow bar, there are not too many segments, and therefore not that many results in total. However, in the $[2^{20}][2^0]$ configurations, the small kernel will need to perform $4 \times total\ elements$ writes to global memory, compared to the $1 \times total\ elements$ needed by the loop-in-map kernel.

I have not included the runtime when using versioned-code explicitly. In both cases the segmented scan approach will be used until there are 2^9 segments, then versioned-code will use the loop-in-map approach for the rest of the configurations. This is a great improvement from always using the loop-in-map approach, but the performance of the segmented scan does not come near the one achieved by my implementation: For the 2^{20} case the segmented scan takes approximately $3000\,\mu s$, and for the 2^{26} case the segmented scan approach takes approximately $44\,000\,\mu s$.

Figure 6.5b shows that for the 2^{26} case, using a low chunking factor for the large kernel has a much more severe impact on performance, when the reduction operator is more complex, and uses a 4-tuple as the operand. We need to perform at least four times as many computations in the group-wide reduction compared to the segmented sum, so when only processing a single element per thread, the total time per element is significantly increased. We can also see that the small kernel has a much higher cost now that we are using the more complex reduction operator, showing that the in-group segmented scan performed by the small kernel has a significant overhead as the number of elements used in the reduction operand grows.

The big picture is the same as for the non-commutative segmented reduction: once there are "enough" segments, using the loop-in-map kernel can give much



(a) Using dataset of total 2^{20} 32-bit floating points.

(b) Using dataset of total 2^{26} 32-bit floating points.

Figure 6.5: Performance for the non-commutative segmented MSS. The horizontal *redomap* line marks the performance of a one-dimensional reduction, *loop-in-map* is the performance for the default loop-in-map strategy, and *thesis* shows the performance when using my implementation to generate code. The yellow vertical bar marks the configuration where the number of segments is equal to the group size for the larger kernel (chunking=1).

---

better runtime. Like before, Figure 6.5 shows that the small kernel is only a good choice in the 2^{20} case.

By also including using the loop-in-map kernel, we can see that my implementation would able to match the performance of the one-dimensional redomap for all configurations of number of segments and segment size.

6.6 Conclusion

We have seen the general idea for how I generate code for segmented reductions and segmented redomaps. we have discussed several optimizations that enable better performance, and suggested other optimizations that could be implemented.

6.6.1 Performance

We have seen that for a simple segmented sum, my implementation outperforms the segmented scan implementation used by the Futhark compiler, both when using a commutative and non-commutative reduction.

We have also seen that when we can use the large kernel with a good chunking factor (in our cases, ≥ 16) we have comparable performance with a one-dimensional reduction. By also using the loop-in-map strategy, we will be able to improve the performance drastically when there are many small segments: for non-commutative reductions we would be able to match the performance of the one-dimensional reduction, for all configurations of number of segments and segment size.

Surprisingly, the small kernel is much faster than the large kernel for the non-commutative reduction of 2^{20} elements; this could suggest that removing the need for padding arrays before transposing them could improve performance for non-commutative reductions significantly.

For computing the segmented MSS with a redomap, we have seen that enabling loop-in-map with my implementation, would greatly outperform Futhark with the versioned-code feature enabled. We have seen that my implementation can match the performance of a non-commutative one-dimensional redomap when the segments are large; however, when there are many segmens, and we need to return multiple values per segment, my current implementation cannot match the performance of the one-dimensional reduction; this could suggest that enabling an optimization to avoid writing any final reduction result that will not be used, can improve the performance significantly.

6.6.2 Dynamically Adjusting Group Size

Dynamically changing the group size, could seem to give a bigger payoff in the Futhark implementation than I initially had expected from the prototype. Specifically, by using a larger group size, we could use the small kernel for more configurations for non-commutative reductions, where the small kernel had the best runtime for the 2^{20} case. For commutative reductions, using a smaller group size would allow us to use the large kernel with a good chunking factor for more configurations.

Chapter 7

Benchmarks

To evaluate the performance of Futhark, benchmarks have been ported to Futhark from Rodinia [7], Accelerate [25], Parboil [29], and FinPar [1]. These benchmarks are available at `https://github.com/HIPERFIT/futhark-benchmarks`[1]

I will use these benchmarks to evaluate the performance improvements from using my implementation for segmented reductions and segmented redomaps. As explained earlier, currently in the Futhark compiler, a segmented reduction will be implemented using a segmented scan, and a segmented redomap will be implemented using the loop-in-map strategy. However, if versioned-code is enabled, code will be generated for both implementing a segmented redomap as a segmented scan and using the loop-in-map strategy, and a runtime decision is made to determine which one to use. Disclaimer: versioned-code is still a work-in-progress, which is why it is not enabled by default in the compiler.

I will compare using the standard Futhark compiler, which I will call *vanilla*, with the Futhark compiler with my implementation for segmented reductions enabled, which I will call *thesis*. I will compare using the Futhark compiler with "versioned code" enabled, which I will call *vanilla+vc*, with the Futhark compiler with both "versioned code" and my implementation for segmented reductions and segmented redomaps enabled, which I will call *thesis+vc*.

When versioned-code is not enabled, I will only look at the benchmarks that uses segmented reductions. When versioned-code is enabled, I will look at the benchmarks that uses either a segmented reduction or a segmented redomap.

First we will look at the speedups we can achieve, and then we will look at the performance of each interesting benchmark in more detail.

For testing the performance I will use the NVIDIA GTX 780 Ti as we have seen throughout the rest of this thesis, but also a NVIDIA GTX Titan Black, a NVIDIA Tesla K40c, and an AMD FireGL 8100.

[1] version I used is from commit id `c5ae760d60749f08968be32c339e49dde3e94b1b`

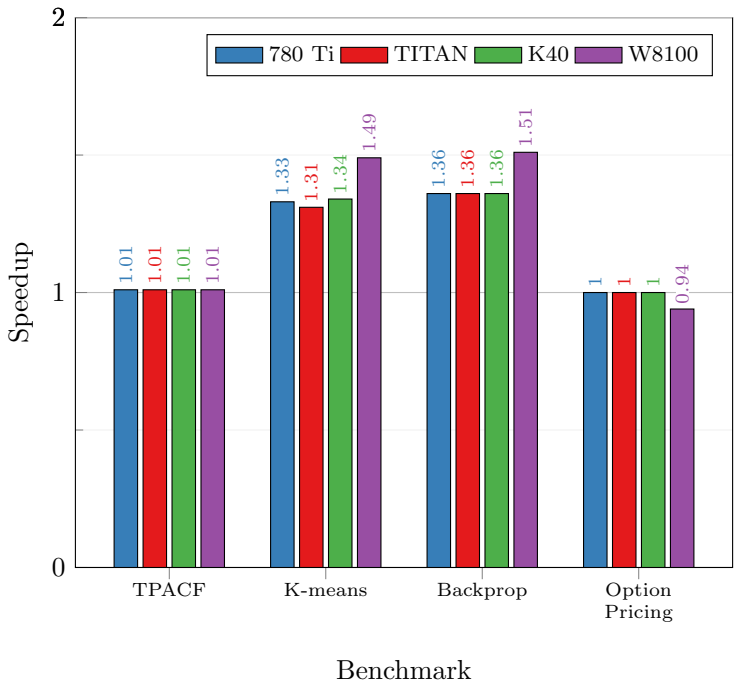


Figure 7.1: Speedups achieved by using my implementation for segmented reductions over vanilla Futhark, on four different GPUs. Only benchmarks that use a segmented reduction are used, and only the largest dataset for each benchmark.

7.1 Speedups

The speedup we will see by using my implementation for segmented reductions and segmented redomaps will depend on the amount of work in a benchmark that is performed by these. If the benchmark only runs a small segmented reduction once, no matter how fast we compute this segmented reduction, we will not see a speedup in the benchmark.

7.1.1 Without Versioned Code

Figure 7.1 shows the speedups achieved by using my implementation for segmented reductions over vanilla Futhark, on four different GPUs. Only benchmarks that use a segmented reduction are used, and only the largest dataset for each benchmark. The used benchmarks are: TPACF from Parboil, K-means from Rodinia, Backprop from Rodinia, and OptionPricing from FinPar[2].

Figure 7.1 shows that we get a speedup on both K-means and Backprop, but

[2]the N-body benchmark from Accelerate does also include segmented redomaps, but I was not able to compile it when I ran my benchmarks.

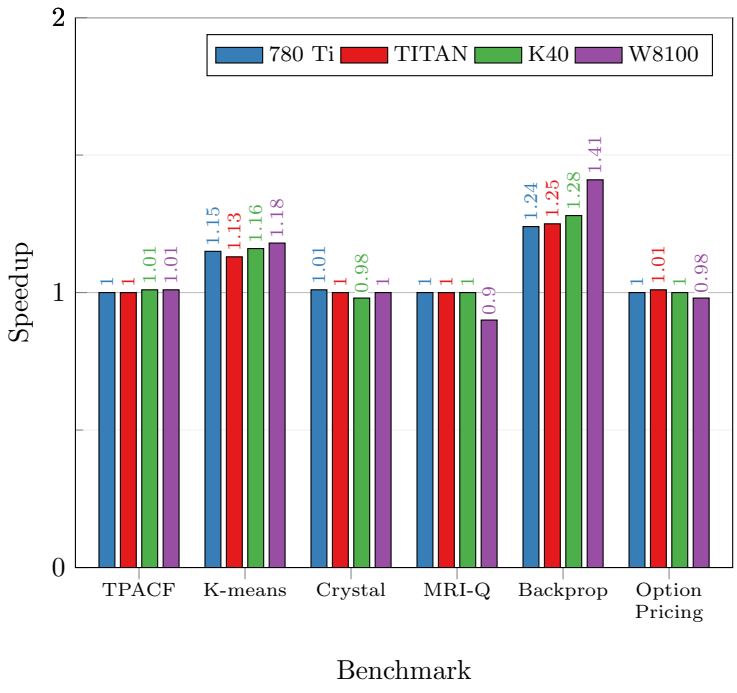


Figure 7.2: Speedups achieved by using versioned code and my implementation for segmented reduction and segmented redomaps over Futhark with versioned code, on four different GPUs. Only benchmarks that use a segmented reduction or a segmented redomap are used, and only the largest dataset for each benchmark.

---

that TPACF and OptionPricing is largely unaffected[3].

We will examine the performance of both K-means more in section 7.3 and Backprop more in section 7.2. TPACF gets a speedup by a harmonic-mean of $1.08\times$ for its smallest dataset, which can be seen in appendix D. I have not looked at either TPACF or OptionPricing in more detail, but their runtime is reported in appendix D.

7.1.2 With Versioned Code

Figure 7.2 shows the speedups achieved by using versioned code and my implementation for segmented reduction and segmented redomaps over Futhark with versioned code, on four different GPUs. Only benchmarks that use a segmented reduction or a segmented redomap are used, and only the largest dataset for each benchmark. We use the same four benchmarks as before, but use additional benchmarks, because we now also handle segmented redomaps. The new benchmarks are Crystal from Accelerate, and MRI-Q from Parboil.

[3]There is a slowdown for OptionPricing on the W8100, but this GPU does sometime behave a bit strange, and there is a large variance in the measured runtimes, so I will not put any emphasis on this.

In Figure 7.2 we can see that the improvements for K-means and Backprop are not as significant as in Figure 7.1, we will look at why this is in section 7.3 and section 7.2. We can also see in Figure 7.2 that the new additions are unaffected by using my implementation for segmented reductions and segmented redomaps[4].

I did not study either Crystal or MRI-Q in further detail, but noted that for Crystal we see a speedup by a harmonic-mean of $1.61\times$ on one of its datasets (dataset #0), which can be seen in appendix D.

7.2 Backprop

Figure 7.3 shows the runtimes for the Backprop benchmark from Rodinia on two datasets, using all four versions of the Futhark compiler, on the four GPUs used for benchmarking.

In Figure 7.3 we can see the reason for the lower speedup when using versioned code than without: the performance is worse when using versioned code, and the absolute improvement made by using my implementation is smaller than without using versioned code. This could be caused by the runtime decision made by the versioned code system for which kernel to use not being optimal; it could also be caused by something unrelated to segmented reductions.

7.2.1 Breakdown of Runtime

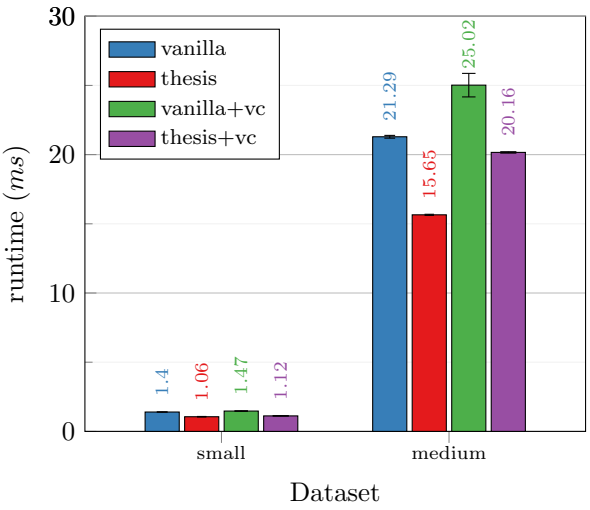
I have tried to quantify how much time is spent computing segmented reductions in the code generated when not using versioned code. Futhark can output a debug summary for how much time have been spent in each kernel[5]. I have used this to compute the time spent in segmented reductions when using the standard Futhark compiler and when using my implementation for segmented reductions (*vanilla* and *thesis* in the figures).

For the standard Futhark compiler I have only counted the time spent in `scan` kernels: this is a generous lower bound on the time taken by the segmented scan implementation, as it does not include the time it takes to produce the flag array, or the time it takes to read out the elements once the scan is done. The reason I did not include these is that it is very hard to distinguish between which map kernels are responsible for these computations. The breakdown can be seen in section D.1

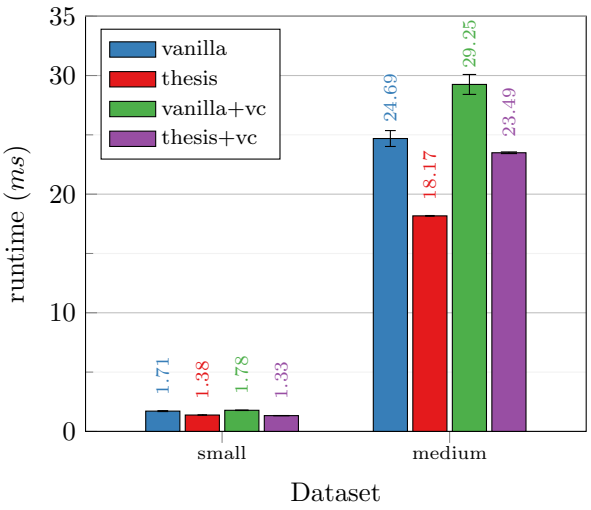
I have only looked at the `medium` dataset on the GTX 780 Ti. For the vanilla version the scans for the segmented scan implementation takes up $4539\,\mu s$ out of the total runtime of $22\,240\,\mu s$ (20.41%). For my implementation, we are using the large kernel with multiple groups per segment, and one group per segment to reduce this recursively. This takes up $363\,\mu s$ of the total runtime of $16\,170\,\mu s$ (2.24%).

[4] There is a slowdown for MRI-Q on the W8100, but this GPU does sometime behave a bit strange, and there is a large variance in the measured runtimes, so I will not put any emphasis on this.

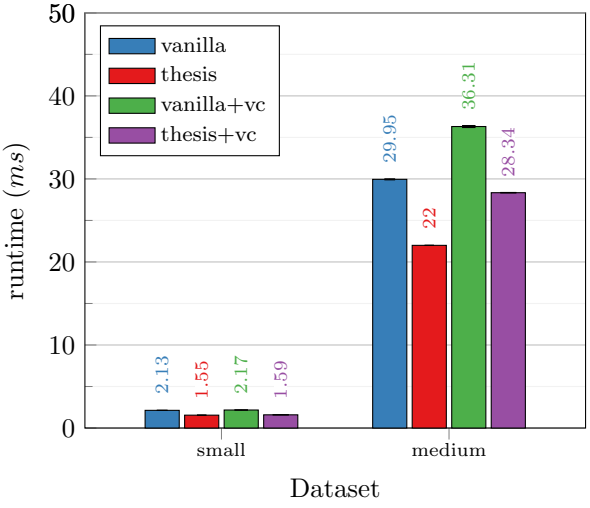
[5] the runtime is increased a bit because some extra synchronization between the host and device is needed.



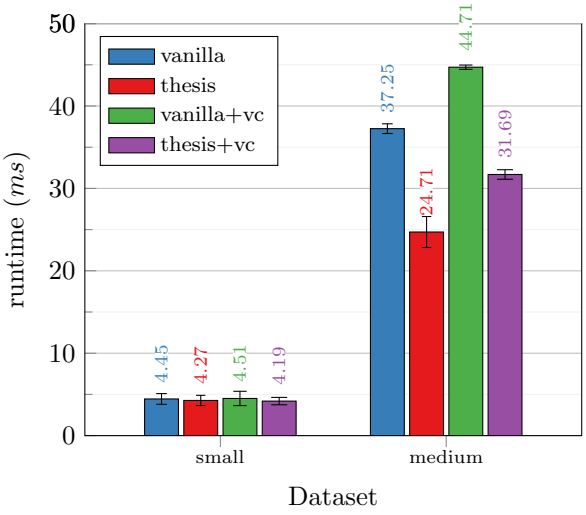
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU



(c) Runtime on the NVIDIA Tesla K40 GPU



(d) Runtime on the AMD FireGL W8100

Figure 7.3: Runtimes for the Backprop benchmark from Rodinia. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.

I am very pleased with these results, as my implementation has successfully made the segmented reduction part of the benchmark much less significant. My intuition is, that it will not be worthwhile trying to improve the performance of the segmented reduction part of the benchmark significantly more than what my implementation has achieved.

I have not looked at all the kernels executed in detail, so it is possible that using my implementation has increased the runtime for other parts of the computation.

7.3 K-means

Figure 7.4 shows the runtimes for the Backprop benchmark from Rodinia on three datasets, using all four versions of the Futhark compiler, on the four GPUs used for benchmarking. The `kdd_cup` dataset was used for the speedup calculations in Figure 7.2 and Figure 7.1.

In Figure 7.4 we can see the reason for the lower speedup when using versioned code than without: using versioned code will improve the runtime. However, using my implementation for segmented reductions will bring the runtime down to the same level, both with and without versioned code enabled.

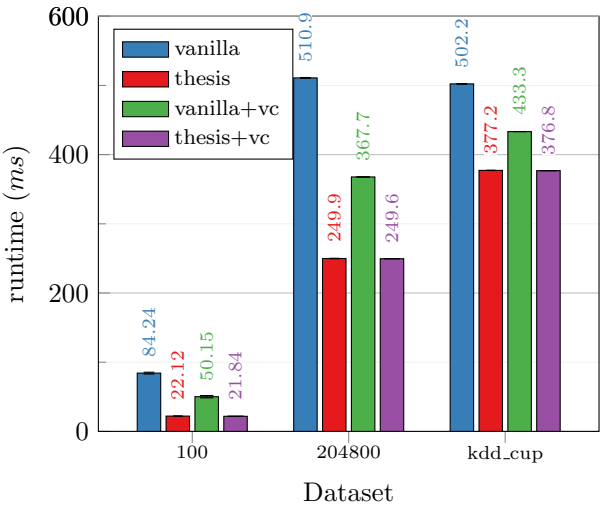
7.3.1 Breakdown of Runtime

As for Backprop, I have tried to quantify how much time is spent computing segmented reductions in the code generated when not using versioned code. As before I looked at how much time was spent in segmented reductions when using the standard Futhark compiler and when using my implementation for segmented reductions (*vanilla* and *thesis* in the figures). For the standard Futhark compiler, I used the same generous lower bound of only counting the time spent in scans. The breakdown can be seen in section D.2.

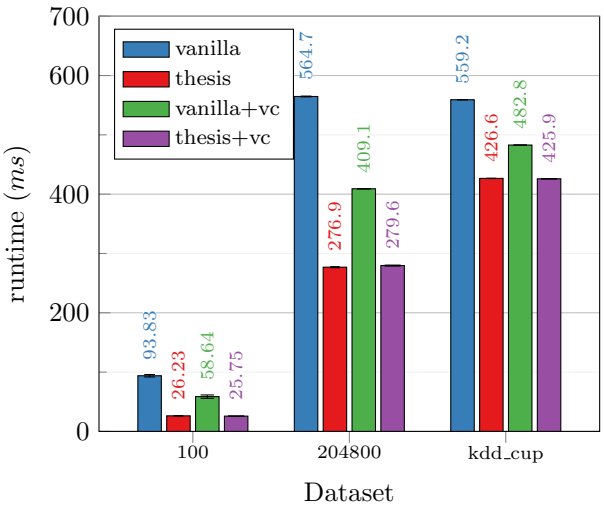
I have only looked at the `kdd_cup` dataset on the GTX 780 Ti. For the vanilla version the scans for the segmented scan implementation takes up $108\,762\,\mu s$ out of the total runtime of $513\,246\,\mu s$ (21.19%). For my implementation, we are using the large kernel with multiple groups per segment, and one group per segment to reduce this recursively. This takes up $5687\,\mu s$ of the total runtime of $400\,487\,\mu s$ (1.1%).

I am very pleased with these results, as my implementation has successfully made the segmented reduction part of the benchmark much less significant. My intuition is, that it will not be worthwhile trying to improve the performance of the segmented reduction part of the benchmark significantly more than what my implementation has achieved.

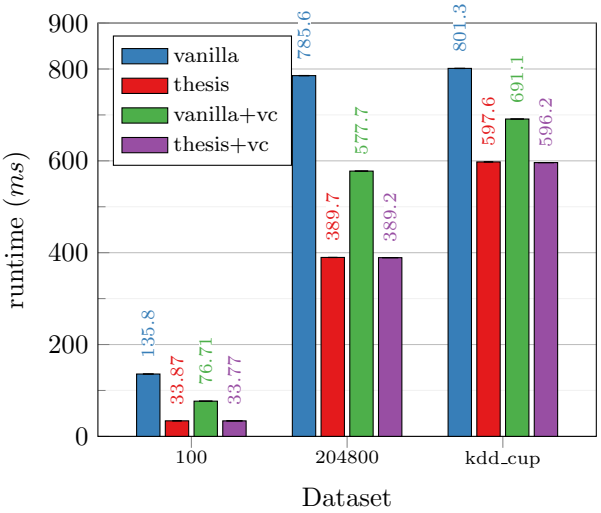
I have not looked at all the kernels executed in detail, so it is possible that using my implementation has increased the runtime for other parts of the computation. For example, I have been able to see there is a difference in the time spent on transpositions (but to the favor of my implementation).



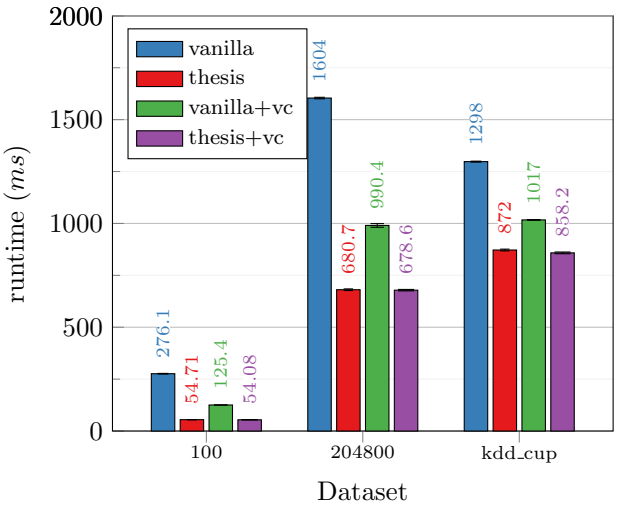
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU



(c) Runtime on the NVIDIA Tesla K40 GPU



(d) Runtime on the AMD FireGL W8100

Figure 7.4: Runtimes for the K-means benchmark from Rodinia. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.

7.4 How to Run These Benchmarks

Instructions for how to compile Futhark in these four versions, and how to run the benchmarks with each version, can be found at https://github.com/RasmusWL/thesis-benchmarks

Chapter 8

Related Work

In this chapter we will look at existing work on computing segmented reductions efficiently on GPUs.

In a very broad sense a segmented reduction is a special case of a histogram computation, where an input array of key-value pairs are processed, to reduce all the values with the same key. This is illustrated in the code below.

```
// 'input_key' and 'input_val' are arrays of size 'n'
// 'res' is a dictionary-like datastructure
for i in 0..n:
    acc = res[ input_key[i] ]
    res[ input_key[i] ] = OP(acc, input_values[i])
```

Some implementations of segmented reductions uses a definition very similar to the histogram computation, with the restriction that a segment is defined as a consecutive range of matching keys. This means there are no restrictions on the size of a segment.

We will call an array where the segment sizes can differ an *irregular segmented array*, or simply say that is has *irregular segments*. An irregular segmented array can be implemented by using a one-dimensional array to hold the elements, and *segment descriptors* to describe the structure of the irregular segments. Segment descriptors are usually implemented by either using an array of keys (as above), or by storing the length of each segment in an array. There are minor variations, such as using a flag array instead of keys, or using offsets for the start of a segment instead of the length of a segment.

An irregular segmented reduction that uses lengths or offsets as its segment descriptors will have to handle the case of a segment having length zero. As this can present some difficulty, some implementations does not allow empty segments.

Problem Difficulty

The problem of computing a segmented reduction for irregular segments is strictly easier than computing a segmented reduction with a histogram: We can

make use of the fact that all elements that needs to be reduced are consecutive in the input array.

In the same sense, computing a segmented reduction for regular segments is strictly easier than for irregular segments: We do not have to worry about how to divide the work evenly, as all segments contain the same number of elements. As we have seen, this also allows us to use different approaches depending on the number of segments and segment size.

Reduction Operator

All of the approaches we will study requires that the reduction operator must be associative; a few approaches allows non-commutative reduction operators, but most require that it is commutative. As we have seen handling non-commutative reduction operators provides a challenge, and it is by no means clear that the solution I have shown in this thesis is optimal. Most approaches require that the neutral element is supplied, and we will look at the downsides of not requiring this.

Compared to Futhark

I have tried to compare the related approaches we will study, to what is possible in Futhark.

First of all I have compared the expressiveness to Futhark's redomap construct. This comes down to two things: (1) if a function can be applied to the input elements before applying the reduction, but without a separate array traversal; and (2) if it is possible to generate output elements in the *same* pass over the input elements as is used for the reduction. Most libraries support the first property, but I have not found any that documents supporting the second property.

It is also instructive to see if multiple reductions can be computed in the same pass over the data or if two separate passes is required. An example could be computing both the minimum and maximum element of an array. A way to support computing multiple reduction in the same pass, is to support using a tuple-of-arrays (structure-of-arrays): This will allow a kernel to access each component efficiently with a memory coalesced access pattern. If only array-of-tuples (array-of-structures) is supported, we are not able to do this.

8.1 NESL – A Baseline Approach

The work on the data parallel programming language NESL, showed how a segmented reduction, capable of handling irregular segment, could be implemented by using a segmented scan, and an instruction to get the last element from each segment. The only requirement is that the reduction operator is associative. This is the approach we explored in subsection 3.4.2.

NESL also shows that a segmented scan can be implemented efficiently by using scans and a flag array [5].

I will consider this way of implementing a segmented reduction as a baseline approach, as it supports irregular segments, and will always fully exploit the available parallelism. This could give it a reasonably fast implementation that will work for any case.

8.2 Accelerate

Accelerate [6] is an array processing language embedded in Haskell. It supports collective array computations such as maps, folds and scans on regular multidimensional arrays. Accelerate has multiple backends to generate code for different targets.

Accelerate supports both segmented reductions on regular arrays with `fold`, and segmented reductions on irregular segmented arrays with `foldSeg`. For the irregular case, the length of each segment is used as the segment descriptor. There is a version of `fold` with the invariant that the arrays are not empty, called `fold1`. Accelerate also supports scans, so it is possible to implement segmented reductions using segmented scans.

There are currently two versions of Accelerate that are of interest: the *old version* published on hackage (version 0.15.1.0) and the *new version* that is still unpublished and currently under development (version 1.0.0.0) The old version uses a backend for generating CUDA coda that is now deprecated (`accelerate-cuda`). The new version generates PTX assembly code for NVIDIA GPUs using LLVM (`accelerate-llvm-ptx`)[1].

It has always been possible to define your own reduction operator, but there has been a significant change between the two version: in the old version the reduction operators had to be both associative and commutative, whereas in the new version it is only required to be associative [25].

To invoke a reduction with the operator `f` over an n-dimensional regular array `xs`, we use `fold f se xs`, where `se` is the starting element for the fold, and does *not* need to be the neutral element of the reduction operator. I believe this is to support a similar semantics to what Haskell programmers are used to from `foldl` and `foldr`. `fold1` does not take a starting element.

Listing 8.1 shows a definition of `sum` for computing the sum of a 1D array, and `segsum` for computing the sum of the inner dimension of a 2D array, for all types `a` that Accelerate recognizes as numbers. The `Acc` type constructor means that the array resides in GPU memory.

Accelerate supports mapping a function to all elements before applying the reduction, in a single pass over the data. Accelerate also applies array-of-tuples to tuple-of-arrays transformation automatically.

[1] it is also possible to use NVIDIA's close source libNVVM to generate code, that have more optimizations; the installed version of LLVM must match the one used by the installed CUDA version

```
import Data.Array.Accelerate as A

sum :: A.Num a => Acc (Array DIM1 a) -> Acc (Scalar a)
sum xs = A.fold (+) 0 xs

segsum :: A.Num a => Acc (Array DIM2 a) -> Acc (Array DIM1 a)
segsum xs = A.fold (+) 0 xs
```

Listing 8.1: Functions for computing sum and segmented sum using Accelerate.

8.2.1 Missing a Neutral Element

Not requiring the starting element `se` to be the neutral element for the reduction, or leaving it out in the case of `fold1`, complicates things for Accelerate:

- For a commutative reduction, where each thread should read multiple elements, a thread will not be able to initialize the accumulator, so extra logic must be added to load the first element.
- For any reduction, if some threads didn't read an element from the input array, a separate function for group reduction that is capable of handling this must be used. Extra logic must also be added to check if some thread didn't read an element.

This illustrates that the semantics of reductions in Futhark helps generate good code for the GPU, because it is required to supply the neutral element. Recognizing when the reduction operator has a neutral element (when it is a member of the `Monoid` class), has been marked as an area of for future work by the Accelerate developers, but is currently not being performed.

8.2.2 Strategy for One-dimensional Reductions

Accelerate handles reduction of 1D arrays using the common approach of multiple steps, where many groups reduce part of the array to produce a number of intermediate values, until there is only one value left.

In the old version, which required a both commutative and associative reduction operator, Accelerate would let each thread read multiple elements with a stride to ensure memory coalescing. We have seen this can increase performance tremendously.

In the new version, which only requires an associative reduction operator, Accelerate does not use the approach we do in Futhark of transpose the input array to allow each thread to read multiple elements with a strided access. Instead, to keep the benefits of processing multiple elements per thread, a slice of the array is assigned to each group, which then executes a number of iterations with the following steps:

1. In iteration i, thread tid reads element $i \times groupsize + tid$ if within bounds.

2. A cooperative reduction is performed within the group. If some thread didn't read an element, a special group-wide reduction is used that can handle this.
3. The results the group-wide reduction in this step is combined with the accumulator in thread 0, or if this is the first iteration the result is simply assigned to the accumulator in thread 0.
4. If there are still more elements to process in the slice assigned to this group, goto step 1.
5. If there is only one group participating in the reduction, this means we will compute the final result now, so if a starting value was supplied (i.e., `fold1` was not used), the starting value is combined with the accumulator.
6. Thread 0 of each group writes its result to global memory.

Currently Accelerate does not recognize when a reduction operator is commutative, and therefore only uses the approach outlined above. It has been marked as an area for future work by the Accelerate developers.

8.2.3 Strategy for Multidimensional Reductions

When Accelerate has to compute a reduction over an array with more than one dimension, it uses the very simple approach of using one group per segment. As we have seen in the prototype (section 5.2) this will not lead to good performance when there are many small segments, or when there a few very large segments. Using one group per segment is also the approach used by Accelerate to handle irregular segments, which can lead to a great imbalance in the workload per group.

8.2.4 Conclusion

As Accelerates uses one group per segment, the implementation presented in this thesis should outperform Accelerate when there is only large segments, and when the segment size is smaller than the number of threads in a group. For commutative reductions, Accelerate uses the same algorithm as for non-commutative reductions, so the implementation presented in this thesis should outperform Accelerate in all configurations of number of segments and segment size.

The approach taken by Accelerate to perform the whole reduction of a segment in one group, could be an improvement over one specific case from my implementation: When dealing with a non-commutative reduction, with a segment size that is slightly larger than the group size, this will result in a small chunking factor (e.g., 2 or 4); Currently my implementation has the choice between two options: (1) performing a transpose, with a higher than usual cost, and gain a small benefit from chunking; or (2) to use the large kernel without chunking, producing a few results per segment that will have to be reduced in

a second step, thereby also incurring the cost of allocating and freeing the array to hold the intermediate results.

Using the approach from Accelerate would improve on the second option, by allowing us to process a whole segment within one group; the performance should the same (or a bit better) than launching multiple groups of the large kernel without chunking. We also get the benefit of removing the cost of allocating and freeing the array for the intermediate results.

8.3 Thrust

Thrust [22] is a C++ library defining common parallel bulk operators such as scan, reduce, and sort. Thrust also supports applying a function to all elements of an array, called a "transform" in Thrust terminology. Thrust has multiple backends, with the CUDA backend being used by default.

Thrust supports both 1D reductions by the function `reduce`, and irregular segmented reductions by the function `reduce_by_key`; not surprisingly, the segment descriptor is an array of keys. Thrust allows you to define your own reduction operator, and requires that it is both commutative and associative. Thrust does not require a neutral element, and only allows for a starting element when computing a 1D reduction. I have not been able to find any documentation for how Thrust implements segmented reductions.

Thrust supports scans, so it is possible to use a segmented scan for performing a segmented reduction with a non-commutative and associative reduction operator.

Thrust has support for "fancy iterators", which can be used to supply the keys to a segmented reduction without allocating any memory. This should cause a significant speedup for a segmented reduction when using regular segments, because the memory bandwidth will only be used for the elements of the input array.

Thrust supports applying a function to all elements of the input array before computing the reduction, which will only require one pass (using the `transform_iterator`). Thrust has support for working with an array-of-tuples as a tuple of arrays, by using the `zip_iterator`.

An example for how to compute a segmented sum over a regular 2D array can be found here.

8.4 CUB

CUB [27] is a C++ library defining common bulk operators much inspired by Thrust. CUB is developed by NVIDIA, and unlike Thrust only targets CUDA, which allows for more tailored customization and optimizations for NVIDIA GPUs.

CUB has support for computing irregular segmented reductions, used either keys or offsets for the segment descriptors. CUB requires that the reduction

operator must be both commutative and associative. An initial value must be supplied, but it is not documented if it should be the neutral element or not.

CUB also supports a very efficient scan implementation, that only requires $2n$ global memory operations for an input array of size n [26]. This should serve as a very good implementation for the baseline approach of using a segmented scan to compute the segmented reduction.

Like Thrust, CUB supports applying a transformation to the input data before computing a reduction. I have not been able to find any support for tuple-of-arrays, this could hurt performance if using the baseline approach of segmented reduction as segmented scan, because we need at least two input arrays: the element array and the flag array.

8.5 Modern GPU

Modern GPU [4] is a CUDA programming library for C++ developed by NVIDIA. One of its main purposes is to serve as a place to learn about high performance CUDA programming by example, as the library has extensive documentation for its algorithms. Many of the algorithms described in Modern GPU are the ones used by CUB; however, due to nature of Modern GPU as a learning platform, its implementation has been kept rather simple.

Modern GPU has an implementation for performing irregular segmented reduce that is well documented. Reduction operators must be both associative and commutative, and it is required to supply the neutral element. It is not possible to apply a function to the input elements before running the reduction; considering the nature of the project as a learning platform, this seems reasonable.

Modern GPU has an interface for either using keys or offsets as the segment descriptor. If empty segments can occur in the input, this must be indicated to the library by setting a boolean flag; then the empty segment descriptors will be filtered out in an initial stage, to make the segmented reduction kernel simpler.

It is possible to prepossess the segment descriptors to perform "segment discovery" as a separate stage. It is claimed to increase throughput when using the offsets as segment descriptors by 10-20%, and give a much larger benefit when using keys as the segment descriptors.

Modern GPU shows that its implementation is much faster than the `reduce_by_key` from Thrust, but it is not stated which version of Thrust was used for this comparison.

8.5.1 Implementation

Modern GPU has the most advanced strategy for handling irregular segments that I have seen described. It assigns an equally sized slice of the input array to each group, where each thread will process multiple consecutive elements.

```
Thread: input vals                    Partials     carry-out
    carry-in
  0: 0   1* 3* 1* 4                   1 3 1        4*            0
  1: 2   4   1* 0  3                  7            3*            4
  2: 0   2   1  5  1                                9             3
  3: 1   3* 4   1  2                  4            7*            12
```

Figure 8.1: Example of how Modern GPU makes a group process a slice of the irregular segmented input array. We use 4 threads with a chunking factor of 5. A star ($*$) marks the end of a segment.

Each thread will load multiple values into local (on-chip) memory using a strided access (to ensure memory coalescing). A thread can then read the consecutive elements it needs from local memory. This use of an in-group on-the-fly transpose of the input data, instead of invoking a separate transpose kernel as we do in Futhark, eliminates $2 \times inputsize$ global memory accesses; however, it limits the chunking factor as all elements processed by a group must fit in local memory.

Each thread will execute the following steps to process its elements, using a number of iterations equal to the chunking factor.

1. Initialize the accumulator to the neutral element.
2. Thread tid will in iteration i read element $tid \times chunking + i$, and calculate the new value for the accumulator by applying the reduction operator.
3. If this was the last element of a segment, the current accumulator will be stored in a list of partial results for this thread, and we will goto step (1). If this was not the last element of a segment, simply goto step (2).
4. Once all iterations has been completed, the current accumulator will be stored as the carry-out value for this thread. A flag value will be set to true if any segments ended in the elements for this thread, and to false otherwise.

An example of the result after executing these steps can be seen in Figure 8.1: There we see how a group of 4 threads, with a chunking factor of 5, processes the following slice of the irregular segmented input array (where $\langle$ means the segment starts at an earlier index, and $\rangle$ means the segment ends at a later index)

$$\left[\ \langle 0,1]\ ,\ [3]\ ,\ [1]\ ,\ [4,2,4,1]\ ,\ [0,3,0,2,1,5,1,1,3]\ ,\ [4,1,2\rangle\ \right]$$

Thread 1 will need to get the carry-out from thread 0 to compute the final result for its first segment, and will need to pass its final accumulator value to thread 3 – because the segment does not end in the values thread 2 "owns". To accomplish this we use an *exclusive* segmented scan of the carry-out values, which will produce the carry-in values show in Figure 8.1. Each thread will have to apply the carry-in values to the first partial result (if that exists). If the

last thread didn't have any segment ending within its elements, it will need to update its accumulator with the carry-in value.

We have now calculated the result for each segment in this slice of the input array, except for the first and last segment of this slice. All the (updated) partial results are written to global memory, as well as the accumulator from the last thread.

We need to combine the result from the carry-out value of a group with the first segment in the next group. The documentation says this is handled by a "(...) special streaming kernel [that] scans carry-out values and redistributes them into the destination reductions".

If there are no segments ending within the slice processed by a group, the on-the-fly transpose will not be applied, and instead of the elaborate segmented reduction described above, a normal group wide reduction will be used (which exploits the commutative property of the reduction operator). It is stated that this optimization increases performance with about 20% when processing large segments. If this optimization was not applied, a non-commutative operator could be used.

The approach taken by Modern GPU will utilize the available parallelism for all types of irregular segments, due to its load balanced approach. The algorithm should be more effective for all cases than the baseline approach of using a global segmented scan.

8.5.2 Conclusion

For commutative reductions with very large segment sizes, the implementation from Modern GPU will use a special kernel making it much more effective. It is unclear to me if there restriction on the maximum chunking factor in this case. If there is not, we can expect the implementation presented in this thesis and the implementation from Modern GPU to have similar performance characteristics. If there is a limitation on the chunking factor for Modern GPU, we can expect the implementation presented in this thesis to have better performance in the case of very large segments.

For reductions with low segment size, I would expect the Modern GPU implementation would beat the small kernel, because of the use of chunking. However, it is unclear to me if the performance of the transpose and loop-in-map kernel will be better than what Modern GPU can achieve.

The core of the Modern GPU algorithm could be applied to compute non-commutative segmented reductions. The case for the small kernel and the loop-in-map kernel is the same as above. For the large non-commutative kernel, is it the same unclear problem as for the loop-in-map kernel: is separate kernels for transpose and reduction better than fusing them together (at the cost of limiting the chunking factor)?

The approach of in-group on-the-fly transpose could be applied to all the kernels presented in this thesis:

- The small kernel could be improved, to be able to load multiple elements per thread, and thereby having several memory transfers in flight at one

time; as we have seen in subsection 5.2.2 even a chunking factor of 4 will improve performance tremendously.

- The loop-in-map kernel could apply the in-group on-the-fly transpose trick when the chunking factor is so low that all elements of a segment will fit into local memory. Compared to performing the transpose in a separate step, this should improve performance a lot.
- When dealing with a non-commutative reduction, the large kernel could adopt the transpose on-the-fly approach to avoid the cost of the separate transpose step completely. As explained above, it is unclear if this would give better performance.

  For arrays that are so small that all elements of a segment will fit into local memory, the in-group on-the-fly transpose trick should bring an improvement in performance compared to both using the large kernel without chunking, and performing the transpose in a separate step.

8.6 PENCIL & PPCG

PENCIL [2] is a "a Platform-Neutral Compute Intermediate Language for Accelerator Programming", which is meant to serve as the target language for compilers of domain specific languages.

PENCIL is a subset of GNU C99 at its core, but has additional language constructs to convey detailed information about the code; for example, invariants can be conveyed by using `__pencil_assume`, the fact that loop iterations has no dependence can be conveyed by using `#pragma pencil independent`, and "summary" functions can be used to let the compiler know what parts of input arrays a library functions is going to be used (without having the source code for the library function).

The authors of PENCIL modified the Polyhedral Parallel Code Generation (PPCG) compiler [30], to be able to compile PENCIL code to efficient OpenCL and CUDA code.

In more recent work, conveying information about reductions was added to the PENCIL language and the PPCG compiler [28]. The implementation allows for custom reduction operators, but they must be both commutative and associative. The user is required to supply the neutral element. Segmented reduction is shown in one of the examples of [28], but it is not described in if any special optimizations are applied to those.

The modified version of PPCG can perform fusion between multiple reductions over the array elements (even if the reductions occur in two different loops). Applying a function to an input element before performing the reduction is allowed. It is not described if producing an output element for each input element in the same pass is supported.

An example for how to compute a segmented sum over a regular array using PENCIL can be seen in Listing 8.2.

```
void zero(float *val){
    *val = 0.0;
}

float addition(float v1, float v2){
    return v1 + v2;
}

void segmented_sum(int M, int N, float data[M][N], float sum[M]){
    for (int j = 0; j < M; j++) {
        __pencil_reduction_var_init(&sum[j], zero);
        for (int i = 0; i < N; i++) {
            __pencil_reduction(&sum[j], data[i][j], addition);
        }
    }
}
```

Listing 8.2: Function for computing the segmented sum of regular array using the PENCIL language.

8.7 Performance Evaluation

In this section we will look at a performance evaluation of some of the related approaches described above.

8.7.1 Optimization Techniques

There are some optimization techniques that are most likely used in highly optimized libraries such as Thrust and CUB, but is current not used in Futhark. The most important of such optimizations for reductions are:

- When loading elements from a global array, it is possible to load multiple elements in one go; for example, instead of loading a single `float`, we can load four elements by using the `float4` vector type. This can give a significant increase in performance, even for a simple copy kernel [23].

  OpenCL supports this feature, but it would require a good deal of extra logic to handle properly: for example, if a segment has an uneven length, we cannot just blindly use `float4`.

- CUDA supports a *shuffle* instruction, that allows threads in the same warp to exchange data without using local (on-chip) memory. This has been shown to be faster than using local memory [9]. It also has the added benefit of reducing the amount of local memory required for the kernel, which can be a limiting factor.

  This feature is currently not supported by OpenCL.

- CUDA supports instructions for atomically updating a global memory location with a set of predefined operations, for example we can atomically add a value to a global memory location with `atomicAdd`. For standard

reduction operators such as sum, product, min, or max, this can help increase performance [24].

Support for specific instructions is dependent on the types of the operand and the compute capability of the GPU: for example, `atomicMin` is not supported for floating point numbers, and `atomAdd` requires compute capability of 6.x or higher to support 64-bit floating point numbers.

Some of the atomic update operations are supported by OpenCL, but for example floating points are not supported.

When comparing performance with highly optimized libraries, we should expect that they will be able to get a better performance if they use an equivalent algorithmic approach.

8.7.2 Segmented Sum

In this section I will compare the performance of my implementation in Futhark for segmented reduction, with the performance of Modern GPU and Thrust. We will compute the segmented sum of 2^{20} and 2^{26} total elements, using different configurations of number of segments and segment size.

Modern GPU

I have implemented a small benchmark using the Modern GPU library. The library is currently in version 2, but I am using the last version 1 release (commit id `1c1cc9e23463bf4e82bad29a3ab34a4ddac99e3d`) because it had a benchmark for (irregular) segmented sum available, and version 2 did not. The newer version might be able to get better performance, but I will use this older version as a baseline.

I modified this benchmark to suit my needs for regular segments, and included it in my GitHub repository along with my tools to run the benchmark from chapter 7. I compiled it using CUDA V7.5.17 with the options `-gencode arch=compute_35,code=sm_35`, as required by the Modern GPU.

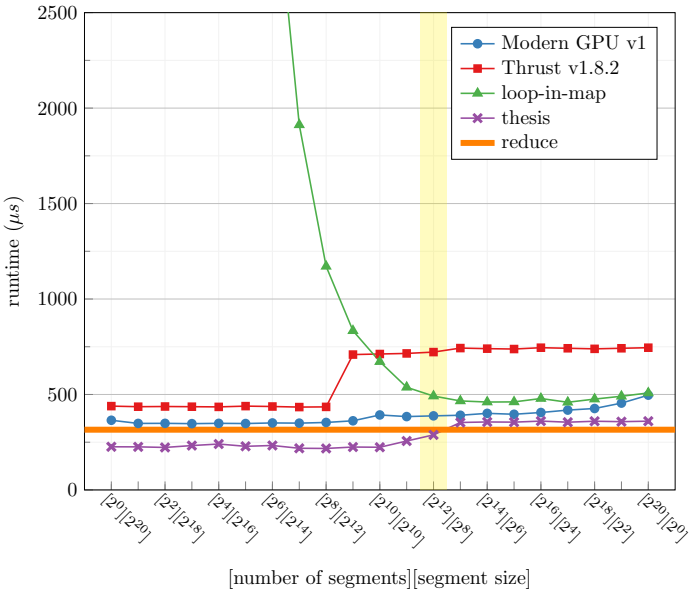
I am using the preprocessed variant of the segmented reductions implemented by Modern GPU; I did see a substantial improvement in performance by doing so (up to $2\times$ speedup when all segments only had a single element).

The reported results are averaged over 10 runs, not including an initial the warmup run.

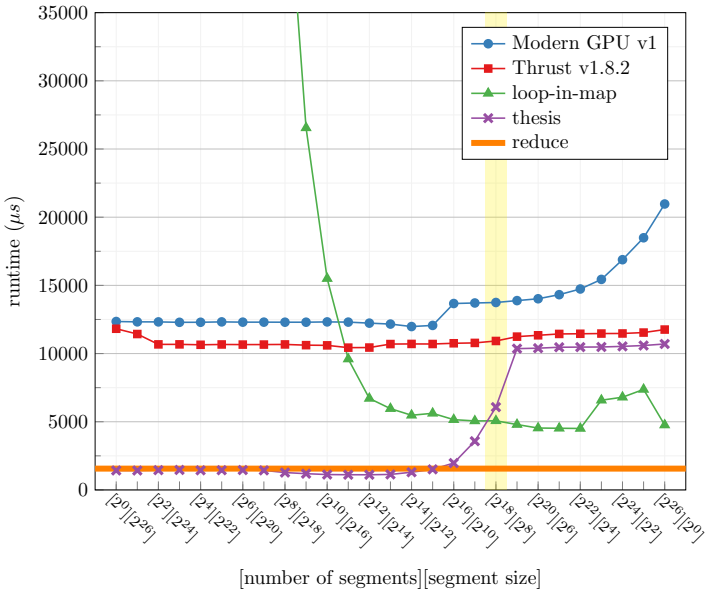
Thrust

As mentioned earlier Thrust has an example available for computing regular segmented reductions, using fancy iterators. I adopted this to compute a regular segmented reduction. I also compiled this using CUDA V7.5.17 with the options `-gencode arch=compute_35,code=sm_35`. The version of thrust used is v1.8.2.

The reported results are averaged over 10 runs, not including an initial the warmup run.



(a) Using dataset of total 2^{20} 32-bit floating points.



(b) Using dataset of total 2^{26} 32-bit floating points.

Figure 8.2: Performance comparison for other implementations of segmented reductions.
As in earlier figures, the horizontal *redomap* line marks the performance of a one-dimensional reduction in Futhark, *loop-in-map* is the performance for the loop-in-map strategy, and *thesis* shows the performance when using my implementation in Futhark to generate code.
The yellow vertical bar marks the configuration where the number of segments is equal to the group size for the larger kernel (chunking=1).
The performance test was run on a GTX 780 ti.

8.7.3 Results

The results of running this performance test on the GTX 780 Ti can be seen in Figure 8.2. I have shown both the performance achieved by my implementation of segmented reductions, and the performance that the loop-in-map implementation gives; in the long run, my implementation should incorporate this strategy as well.

In Figure 8.2a we can see that for the 2^{20} case, both Modern GPU and Thrust are only a bit slower than my implementation when there are less than 2^8 segments. At this point, the runtime of Thrust jumps up, and stays flat for the rest of the configurations; I do not have any idea why this happens. We can see that the performance of Modern GPU stays fairly flat for the remainder of the configurations, so its load balancing scheme is working good in this case.

In Figure 8.2b we can see that for the 2^{26} case, the performance of Modern GPU and Thrust are fairly similar when there are fewer than 2^{15} segments, but they are nowhere near the runtime of $\approx$1500 µs that the one dimensional reduction and my implementation can achieve. As the number of segments

increases, the performance of the Modern GPU implementation suffers more and more; I do not have a good idea about why this happens. We can see that the runtime of the Thrust implementation is fairly constant over all configurations of number of segments and segment sizes. My small kernel has a runtime that is only slightly better than the Thrust implementation, but as we can use the loop-in-map for these configurations, we will still have a significantly better final runtime.

8.7.4 Conclusion

We have seen that the Thrust and Modern GPU implementations for segmented reductions have a fairly competitive runtime, over all configurations of number of segments and segment size, when there are 2^{20} elements. I would assume that this would also be the case when there are fewer elements.

However, when dealing with 2^{26} elements, there is no question that my implementation has a much better performance. When we can use the large kernel with a good chunking factor (≥ 8) we achieve a speedup of approximately $8\times$. As the number of segments increases, we will change to the loop-in-map kernel; we don't have as superior performance, but still achieve a speedup of at least $2\times$, except for the cases where our transpose cost is higher.

Chapter 9

Conclusions and Future Work

In this thesis we have explored how to generate efficient code for segmented reductions and segmented redomaps for GPUs. The initial starting point for the Futhark compiler was that segmented reductions would be computed using a segmented scan, and that segmented redomaps would be computed by letting a single thread compute the redomap for a whole segment sequentially.

Throughout this thesis we have reached the following conclusions:

- The experiments on the prototype implementation for a segmented sum, showed us how using three different strategies for computing segmented reductions are applicable to different configurations of number of segments and segment size. The *large* kernel is effective on large segments; the *loop-in-map* kernel is effective when are many segments, as it uses a single thread to process a segment; and the *small* kernel can be used in the remaining cases, and is an important part of the puzzle although it did not perform best in the experiments we investigated.

  We have seen how reading multiple elements per thread significantly increases performance of the large kernel; as transposition is required for non-commutative reductions to use this technique, commutative and non-commutative segmented reductions have different performance characteristics.

  We have devised a simple algorithm to decide which kernel to use for a given configuration of number of segments and segment size. We based this decision on the simple tuning parameters of groups size and how many threads are needed to fully utilize the hardware. We have seen that this algorithm does not always make the optimal choice, and argued that the optimal choice is heavily based on whether the reduction is commutative or non-commutative; the complexity of the reduction operator used; the configuration of number of segments and segment size; and the GPU device being used. We have discussed additional tuning parameters that could allow an autotuning project to optimize the decision algorithm, for a specific instance of a segmented reduction.

  Finally, we saw how the bandwidth of the GPU device is a limiting factor for how fast a segmented reduction can be computed; there is a large

difference between the best possible runtime when using one segment with n elements, and n segments with one element.

- My implementation in the Futhark compiler, for code generation for segmented reductions, achieves better performance, compared to the segmented scan used by the unmodified Futhark compiler, for the problems we have studied. We have seen that when we can use the large kernel with a good chunking factor (i.e., ≥ 16), we have comparable runtime with a one-dimensional reduction; By also using the loop-in-map strategy, we will be able to improve the performance drastically when there are many small segments: for non-commutative reductions we would be able to match the performance of the one-dimensional reduction, for all configurations of number of segments and segment size.

  For segmented redomaps, we have studied the maximum segment sum problem, which uses a non-trivial reduction operator. We have seen that enabling loop-in-map with my implementation, would greatly outperform the implementation in Futhark with versioned-code enabled. We have seen that my implementation can match the performance of a non-commutative one-dimensional redomap when the segments are large; however, when there are many segments, and we need to return multiple values per segment, it becomes impossible to match the performance of the one-dimensional reduction.

- We have seen that my implementation can improve the performance of ported benchmarks from the Rodinia and Parboil benchmark suites. However, for some benchmarks that use segmented reductions or segmented redomaps, the amount of work performed by the segmented reductions/redomaps is so insignificant that my implementation leads to no performance improvement.

  From evaluation on four different GPUs, we have seen a large improvement in runtime by using my implementation on the K-means and Backprop benchmarks from Rodinia. Compared to the unmodified Futhark compiler, my implementation achieved speedups by a harmonic-mean factor of $1.39\times$ for Backprop, reducing the percentage of runtime spent in segmented reductions from at least 20.41% to 2.24%; and we achieved a speedup by a harmonic-mean factor of $1.36\times$ for K-means, reducing the percentage of runtime spent in segmented reductions from at least 21.19% to 1.1%.

- Finally, we have seen that not many other GPU libraries or compilers have developed an implementation for segmented reductions of regular arrays, that is as efficient as the one presented in my thesis.

  GPU libraries such as Thrust, CUB, and Modern GPU, supports irregular segmented reductions, but seems to have no special support for regular segmented reductions; this means that they are solving a more general problem, and cannot exploit the regularity of multidimensional arrays. Furthermore, these GPU libraries only support commutative reduction operators; if we need to compute a non-commutative reduction, we need to use a segmented scan, which we have seen is not as efficient.

Accelerate, an array processing language embedded in Haskell, has an implementation for computing a segmented reduction over regular multidimensional arrays. However, it always uses a single group to process a segment, so this implementation will only be efficient when there are enough segments, and the segments are large enough, to fully utilize the hardware. Accelerate supports both commutative and non-commutative reduction operators, but uses the exact same code to execute both types; as we have seen in both the prototype and the evaluation of my implementation, treating commutative and non-commutative reductions separately, can lead to huge performance improvements.

We have seen a simple performance evaluation using segmented sum, comparing my implementation with Thrust and Modern GPU. We have seen that when using 2^{20} total elements, there was not a large difference in performance, but when using 2^{26} elements, my implementation was much faster for all configurations of number of segments and segment size; specifically when there were only few very large segments, my implementation outshine the other and achieved a speedup of approximately $8\times$.

9.1 Future work

To further improve the code generation of segmented reductions and segmented redomaps, we could explore the following subjects in more detail:

- In this thesis we have only used powers of two for the number of segments and the segment size. I have a clear conviction that the approaches and the lessons learned will be applicable to using any number of segments and any segment size, but it would make sense to verify this.

  Likewise, we have mostly looked at the performance for 32-bit floating points. I don't see any reason why the work in this thesis should be invalidated by using a different datatype, but it would make sense to verify this.

- We could try to adopt the notion of an in-group on-the-fly transpose as used by Modern GPU. As discussed in subsection 8.5.2, this technique could be applied to all three kernels:

  - For the large and loop-in-map kernel, this approach would eliminate the separate transpose kernel, and therefore $2 \times inputsize$ global memory accesses; however, this approach would limit the maximum chunking factor as all elements processed by a group must fit in local memory.
  - For the small kernel, we could use this approach to enable it to use chunking. As we have seen throughout this thesis, using chunking can improve performance significantly.

- We could explore other ways to implement the small kernel, that does not rely on an in-group segmented scan. My intuition is that by using a flag

array as the segmented scan does, we can make a flag-based segmented reduce within a group; this will allow us to skip the second step of a segmented scan where the intermediate results are redistributed, and reduce the number of reads and writes to local memory.

It might also be possible to avoid using the flag array completely, and rely on index calculations to figure out if the value from two threads should be combined or not. It is unclear to me which of these approaches would be better.

- We could use separate group-wide reduction kernels for commutative and non-commutative reductions. The fully optimized commutative reduction kernel from NVIDIA, mentioned in subsection 4.3.2, did get significantly lower runtime in my preliminary performance experiment (see appendix A).

- We could try to create an alternate version of the loop-in-map kernel, that uses multiple threads to compute multiple intermediate results per segment. As all the intermediate results reside within the same group, we can simply apply the same technique as in the small kernel to reduce these to a single result per segment; thereby we eliminate the need for allocating a temporary array, and the global memory accesses to read and write from it.

  Using this technique would enable us to use the loop-in-map kernel on a wider range of configurations of number of segments and segment sizes. My intuition is that this would be most effective for non-commutative reductions when a transpose would be required anyway.

- We could implement a non-chunking version of the large kernel in Futhark, to allow completely avoiding the cost of a transpose for non-commutative reductions. It is unclear to me in which cases this would be the right thing to do, so this will need to be explored in depth.

- We could work on the part the Futhark compiler responsible for performing transpositions and padding arrays. Currently when an array is transposed because we will use chunking, we will always perform a memory copy and then perform the transpose. As mentioned in section 6.4 it should be possible to avoid the cost of performing padding, even if the array size is not a multiple of the number of threads that will be used.

- We could work on an autotuning project to automatically tune the decision algorithm to a specific problem, on a specific GPU. We could also try to investigate a wider range of segmented reductions and segmented redomaps, to allow us to come up with better heuristics.

  I would imagine my idea of adjusting the group size dynamically from subsection 5.4.2 could also fall under such an autotuning project.

- When using the large kernel, and it is reasonable to use multiple groups per segment, it might be beneficial to start by increasing chunking instead; the data on chunking in subsection 5.2.2 showed that even a chunking factor of 4 can increase performance significantly. When faced with a large segment, we could start by increasing the chunking factor until it reaches

a limit, and only then start using multiple groups per segment. This limit would be yet an other tuning parameter; we could try different heuristics what works best in general, or let an autotuning project control the value.

Chapter 10

Acknowledgments

I would like to thank Cosmin and Troels for their guidance and support throughout this project.

I would also like to thank Marianne, Sofia, Kasper, René, and Alexander for their insightful suggestions to making the thesis more clear, and their help with proofreading.

I would like to thank Trevor L. McDonell, one of the Accelerate developers, for clarifying how Accelerate computes segmented reductions.

References

[1] Christian Andreetta et al. "FinPar: A Parallel Financial Benchmark". In: *ACM Trans. Archit. Code Optim.* 13.2 (June 2016), 18:1–18:27. ISSN: 1544-3566. DOI: 10.1145/2898354.

[2] Riyadh Baghdadi et al. "PENCIL: A platform-neutral compute intermediate language for accelerator programming". In: *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE. 2015, pp. 138–149.

[3] Erik Barendsen and Sjaak Smetsers. "Conventional and uniqueness typing in graph rewrite systems". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1993, pp. 41–51.

[4] Sean Baxter. *Modern GPU 1.0*. 2013. URL: https://moderngpu.github.io/segreduce.html.

[5] Guy E Blelloch. "Scans as primitive parallel operations". In: *IEEE Transactions on computers* 38.11 (1989), pp. 1526–1538. URL: http://people.eecs.berkeley.edu/~driscoll/cs267/papers/scan_primitive.pdf.

[6] Manuel MT Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14.

[7] S. Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Oct. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[8] *CUDA C Best Practices Guide v8.0 (January 12, 2017)*. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

[9] Julien Demouth. "Kepler's SHUFFLE (SHFL): Tips and Tricks". In: *Proceedings of the GPU technology conference, GTC*. 2013. URL: http://on-demand.gputechconf.com/gtc/2013/presentations/S3174-Kepler-Shuffle-Tips-Tricks.pdf.

[10] Martin Elsman and Martin Dybdal. "Compiling a Subset of APL Into a Typed Intermediate Language". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. ACM, 2014, 101:101–101:106. URL: http://hiperfit.dk/pdf/array14_final.pdf.

[11] *Futhark User's Guide*. URL: https://futhark.readthedocs.io/en/latest/.

[12] *Futhark's Official Webpage*. URL: http://futhark-lang.org/.

[13] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

[14] Mark Harris. "An Efficient Matrix Transpose in CUDA C/C++". In: 2013. URL: https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/.

[15] Mark Harris. "Optimizing Parallel Reduction in CUDA". In: (2007). URL: http://www.dps.uibk.ac.at/~cosenza/teaching/gpu/reductionHarris.pdf.

[16] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel prefix sum (scan) with CUDA". In: (2007), pp. 851–876. URL: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html.

[17] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. "Size Slicing: A Hybrid Approach to Size Inference in Futhark". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC '14. ACM, 2014, pp. 31–42. ISBN: 978-1-4503-3040-4. URL: https://futhark-lang.org/publications/fhpc14.pdf.

[18] Troels Henriksen, Ken Friis Larsen, and Cosmin E Oancea. "Design and GPGPU performance of Futhark's redomap construct". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ACM. 2016, pp. 17–24. URL: https://futhark-lang.org/publications/array16.pdf.

[19] Troels Henriksen and Cosmin E. Oancea. "Bounds Checking: An Instance of Hybrid Analysis". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. ACM, 2014, 88:88–88:94. URL: https://futhark-lang.org/publications/array14.pdf.

[20] Troels Henriksen and Cosmin Eugen Oancea. "A T2 graph-reduction approach to fusion". In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM. 2013, pp. 47–58. URL: https://futhark-lang.org/publications/fhpc13.pdf.

[21] Troels Henriksen et al. "APL on GPUs: A TAIL from the Past, Scribbled in Futhark". In: *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. FHPC 2016. ACM, 2016, pp. 38–43. URL: https://futhark-lang.org/publications/fhpc16.pdf.

[22] Jared Hoberock and Nathan Bell. *Thrust: A Parallel Algorithms Library (v1.8.1)*. URL: http://thrust.github.io/.

[23] Justin Luitjens. *CUDA Pro Tip: Increase Performance with Vectorized Memory Access*. 2013. URL: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/.

[24] Justin Luitjens. *Faster Parallel Reductions on Kepler*. 2014. URL: https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[25] Trevor L McDonell. "Optimising Purely Functional GPU Programs". PhD thesis. University of New South Wales, July 2014. URL: http://www.cse.unsw.edu.au/~tmcdonell/papers/TrevorMcDonell_PhD_submission.pdf.

[26] Duane Merrill and Michael Garland. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Tech. rep. 2016. URL: https://research.nvidia.com/sites/default/files/publications/nvr-2016-002.pdf.

[27] Duane Merrill and NVIDIA. *CUB: A Flexible Library of Cooperative Thread-block Primitives (v1.6.4)*. URL: http://nvlabs.github.io/cub/.

[28] Chandan Reddy, Michael Kruse, and Albert Cohen. "Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel: ACM, 2016, pp. 87–97. ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967950. URL: http://doi.acm.org/10.1145/2967938.2967950.

[29] John A Stratton et al. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* 127 (2012).

[30] Sven Verdoolaege et al. "Polyhedral parallel code generation for CUDA". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), p. 54.

[31] Vasily Volkov. "Better performance at lower occupancy". In: *Proceedings of the GPU technology conference, GTC*. Vol. 10. San Jose, CA. 2010, p. 16. URL: http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf.

Appendices

Appendix A

Performance of CUDA reductions

The data in this very crude table comes from a GeForce GTX 780 Ti, and is averaged over 100 runs. "nvidia comm" is an implementation for the most efficient version from [15], that only works for a commutative reduction operator.

All reductions compute the sum of the input array, which are 32-bit floating points (result is verified to be correct, within a margin of error).

There is an upper limit of the size of the grid in the x- and y-dimension of $2^{16}-1 = 65535$. For these simple kernels that only use blocks in the x-dimension, this limits us launching at most $2^{16} - 1$ blocks.

I only used powers of two for the number of elements, so the largest test case for 2^{25} elements was also able to be run using a block size of 1024.

```
Device : GeForce GTX 780 Ti
n=1024, blockSize=1024, num_blocks=1
                  Routine             Microseconds
                version 1                8.28
                version 2                4.75
              nvidia comm                4.46

n=1024, blockSize=512, num_blocks=2
                  Routine             Microseconds
                version 1                6.26
                version 2                4.40
              nvidia comm                4.36

n=1024, blockSize=256, num_blocks=4
                  Routine             Microseconds
                version 1                5.22
                version 2                4.41
              nvidia comm                4.45

n=1024, blockSize=128, num_blocks=8
                  Routine             Microseconds
                version 1                4.79
                version 2                4.42
              nvidia comm                4.40
```

```
n=1048576, blockSize=1024, num_blocks=1024
                  Routine             Microseconds
                version 1              221.78
                version 2               86.25
              nvidia comm               72.56

n=1048576, blockSize=512, num_blocks=2048
                  Routine             Microseconds
                version 1              165.66
                version 2               79.44
              nvidia comm               57.94

n=1048576, blockSize=256, num_blocks=4096
                  Routine             Microseconds
                version 1              128.76
                version 2               79.76
              nvidia comm               49.48

n=1048576, blockSize=128, num_blocks=8192
                  Routine             Microseconds
                version 1               93.32
                version 2               82.11
              nvidia comm               41.65

n=33554432, blockSize=1024, num_blocks=32768
                  Routine             Microseconds
                version 1             6021.89
                version 2             2350.90
              nvidia comm             1926.94
```

Appendix B

Prototype Raw Performance for all Group Sizes

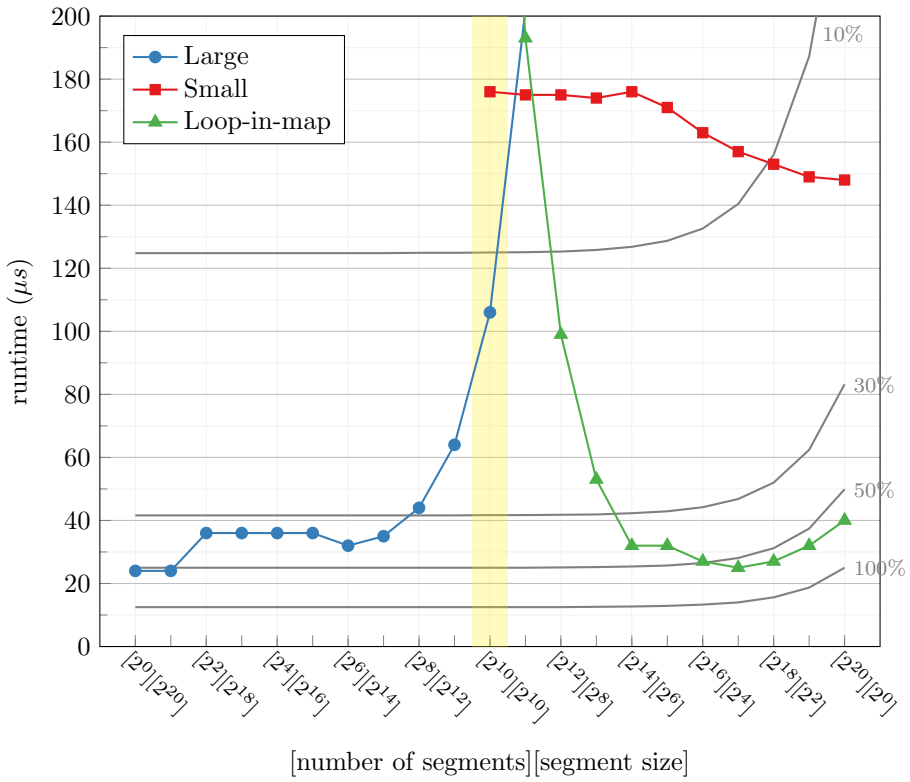


Figure B.1: Using groupsize 1024 on 2^{20} 32-bit floating points.

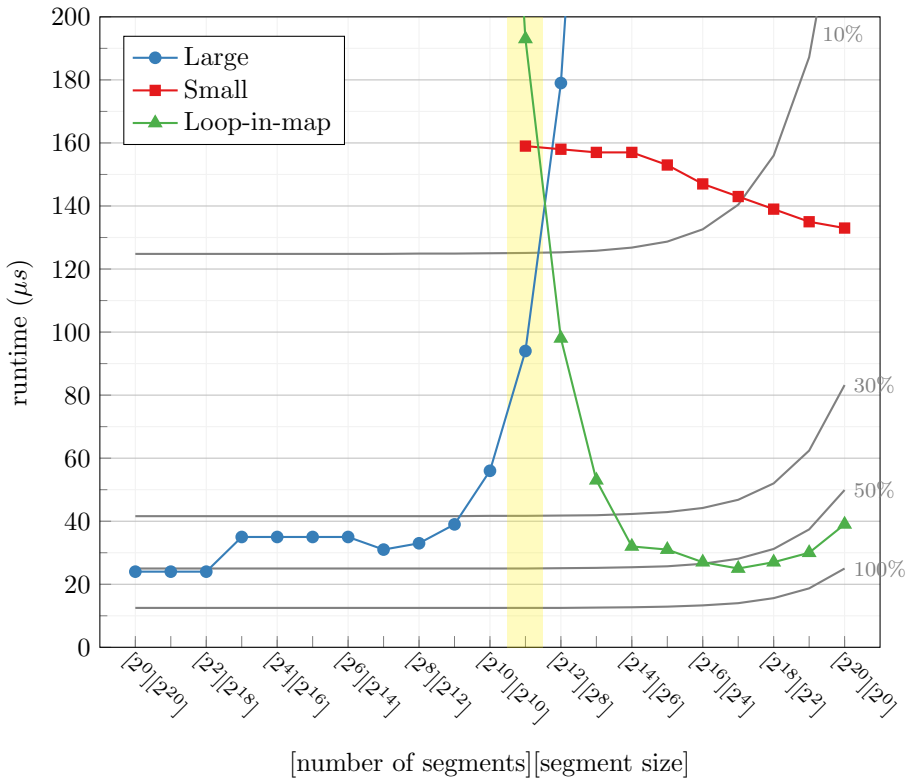


Figure B.2: Using groupsize 512 on 2^{20} 32-bit floating points.

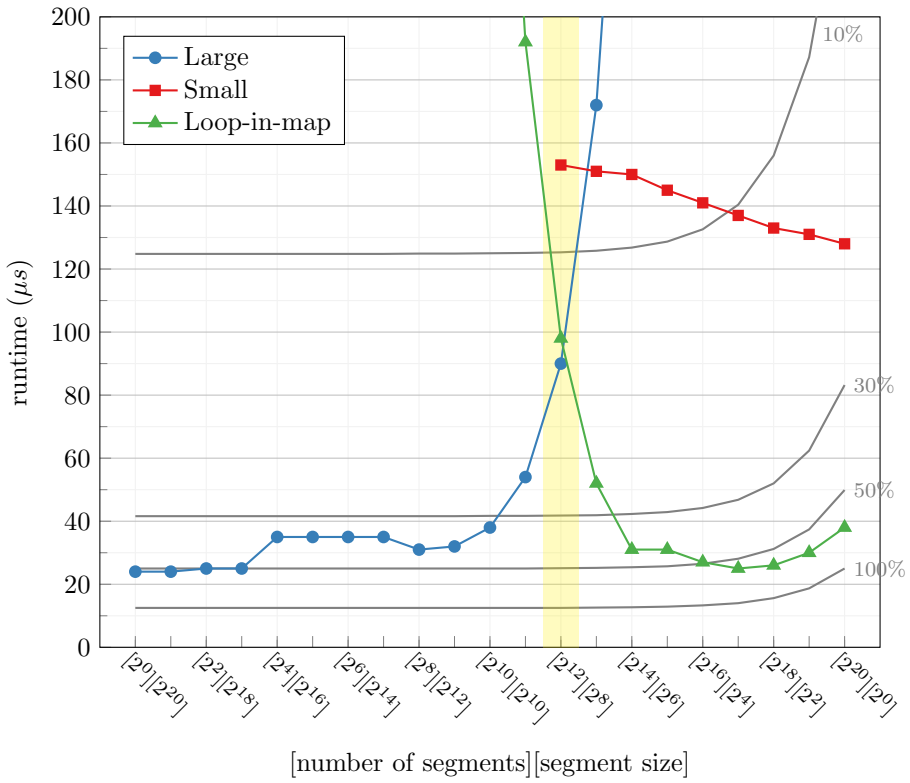


Figure B.3: Using groupsize 256 on 2^{20} 32-bit floating points.

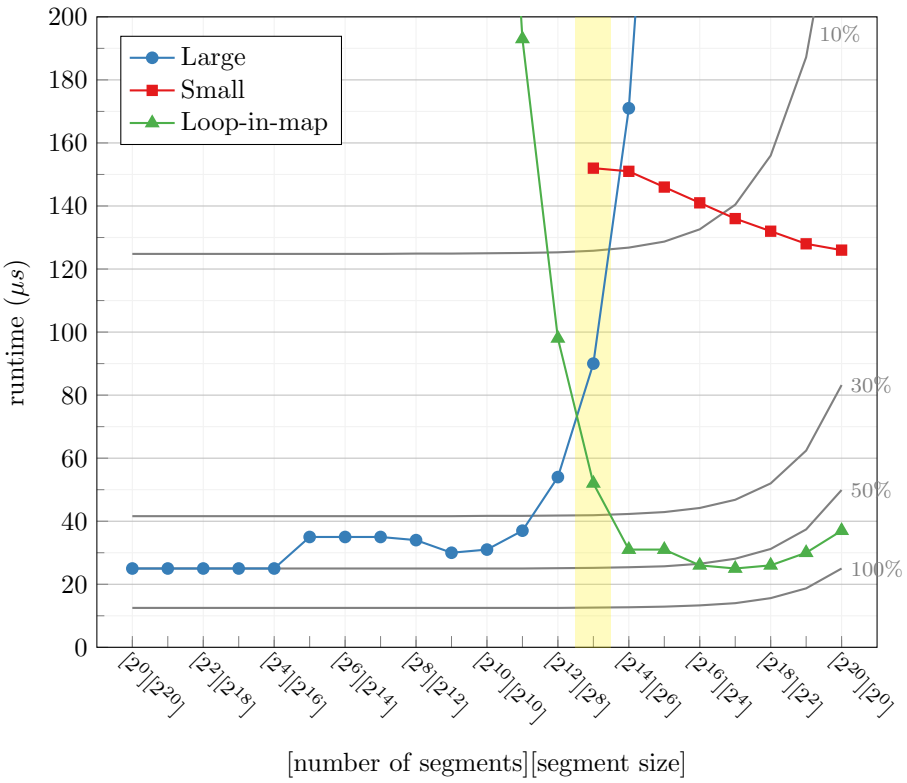


Figure B.4: Using groupsize 128 on 2^{20} 32-bit floating points.

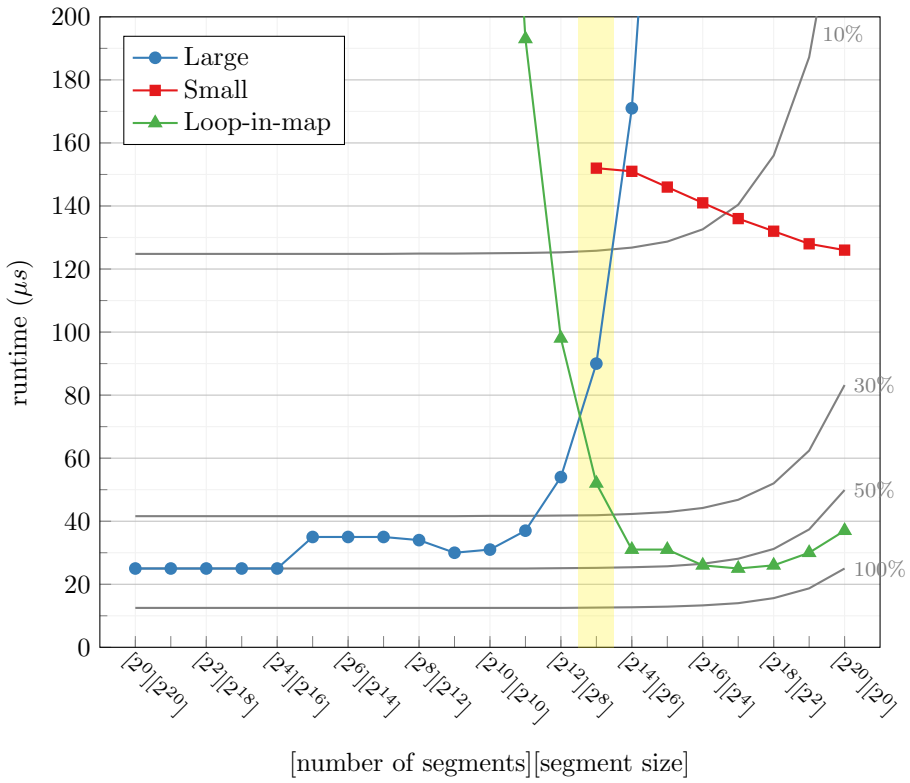


Figure B.5: Using groupsize 1024 on 2^{26} 32-bit floating points.

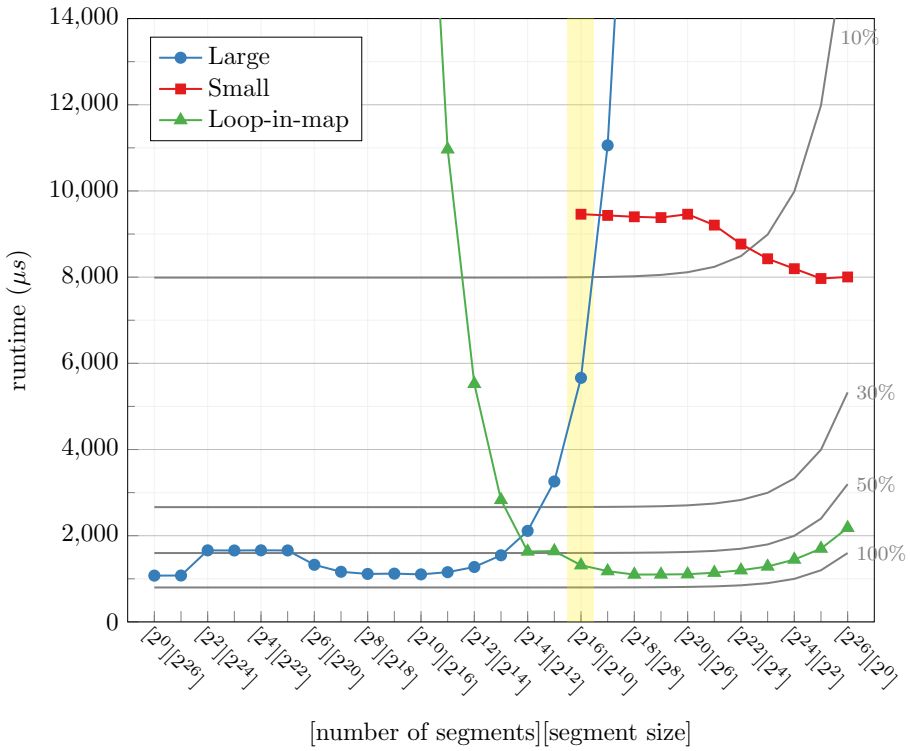


Figure B.6: Using groupsize 512 on 2^{26} 32-bit floating points.

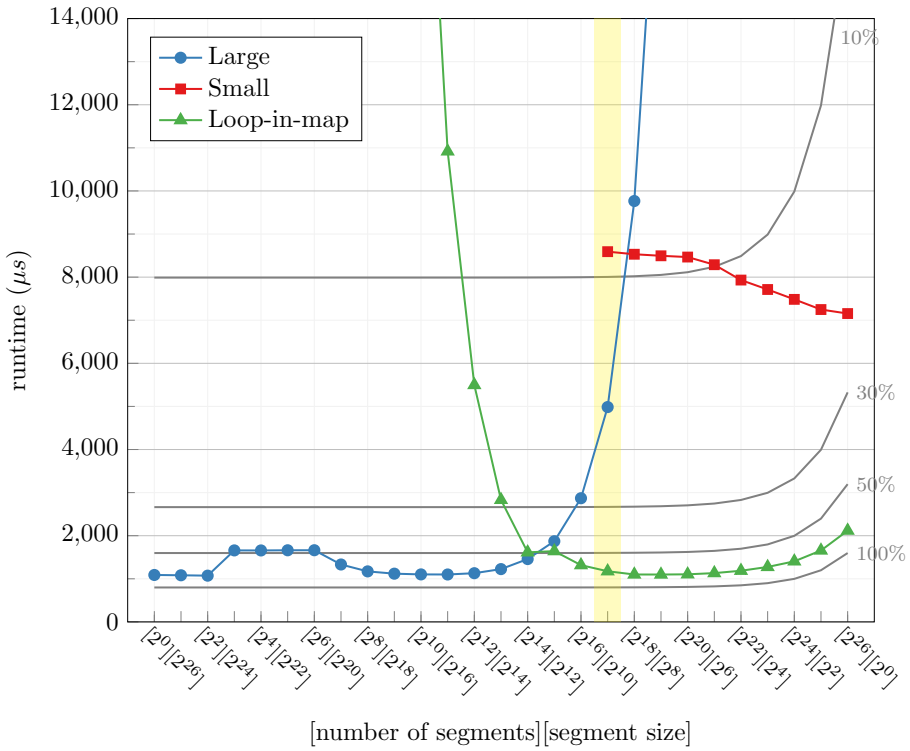


Figure B.7: Using groupsize 256 on 2^{26} 32-bit floating points.

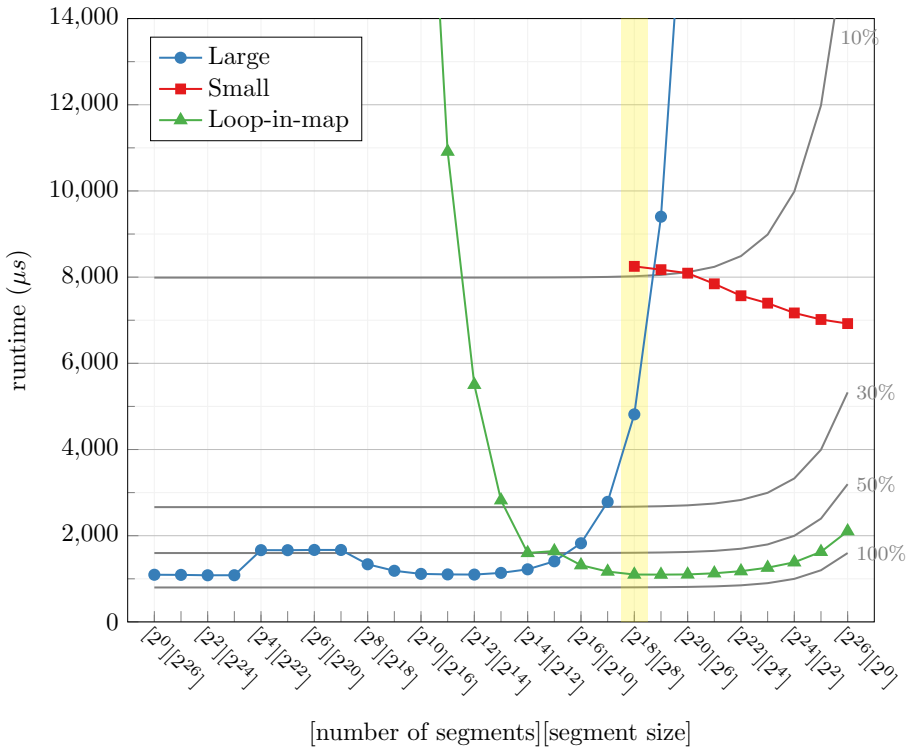


Figure B.8: Using groupsize 128 on 2^{26} 32-bit floating points.

Appendix C

New Transpose Kernel

I created a new transpose kernel to handle the cases where either dimension is smaller than the tile size. In the case of a low height of the input array, the new transpose kernel will read $m \times tilesize$ elements of each row instead of just $tilesize$, where $m = \lfloor \frac{tilesize}{height} \rfloor$.

The new transpose kernel should only be used in the cases where $m \geq 2$, which covers the edge cases from the "normal" transpose kernel (the one discussed in subsection 4.3.3)

An important detail for this new transpose is that it does not help in the case where the height falls with in $\frac{tilesize}{2} < height < tilesize$, because we cannot use $m \geq 2$. We know that the percentage of active threads in these cases must be strictly greater than 50%.

My implementation for the Futhark compiler can be seen at GitHub.

C.1 Example

If we use a `TILE_DIM` of 4, and have a $[2][8]$ array

```
[ [ 0,  1,  2,  3,  4,  5,  6,  7]
, [ 8,  9, 10, 11, 12, 13, 14, 15]
]
```

normally we would use 2 tiles to transpose this (one for the first four element of both rows, and one for the last four of both rows). If we consider the number of elements the current would process per tile (`height * TILE_DIM`), we should only launch the "low height" kernel if it has room for all the elements that 2 normal tiles would process.

This means that a $[3][16]$ array will not benefit from the "low height" kernel when using `TILE_DIM` of 4, and we should launch the normal transpose kernel.

It might be possible to create an other kernel that can handles the cases where `TILE_DIM > height > TILE_DIM/2`, but this kernel does not.

C.2 Implementation

The new kernel for handling a low height of the input array can be seen below

```
// Will consume `mulx * TILE` elements of each row of the input
    array, thereby
// making up for the lack of height in the input array
template <int TILE>
__global__ void transpose_new_lowheight(const float* A, float* B,
    int heightA, int widthA, int mulx) {

    __shared__ float tile[TILE][TILE+1];

    int x = blockIdx.x * TILE * mulx + threadIdx.x + (threadIdx.y %
         mulx)*TILE;
    int y = blockIdx.y * TILE + (threadIdx.y / mulx);

    int ind = y * widthA + x;
    if( x < widthA && y < heightA ) {
        tile[threadIdx.y][threadIdx.x] = A[ind];
    }
    __syncthreads();

    x = blockIdx.y * TILE + (threadIdx.x / mulx);
    y = blockIdx.x * TILE * mulx + threadIdx.y + (threadIdx.x %
        mulx)*TILE;

    ind = y * heightA + x;
    if( x < heightA && y < widthA ) {
        B[ind] = tile[threadIdx.x][threadIdx.y];
    }
}
```

How we should launch it is illustrated below

```
if (height <= TILE_DIM/2 && width > TILE_DIM) {
    int mulx = TILE_DIM / height;
    int newwidth = (width+mulx-1)/mulx;

    dim3 dimGrid ((newwidth+TILE_DIM-1)/TILE_DIM, (height+TILE_DIM
        -1)/TILE_DIM, 1);
    dim3 dimBlock(TILE_DIM, TILE_DIM, 1);

    transpose_new_lowheight<TILE_DIM> <<<dimGrid, dimBlock>>>(
        d_idata, d_odata, height, width, mulx);
}
```

The new kernel for handling a low width of the input array can be seen below

```
template <int TILE>
__global__ void transpose_new_lowwidth(const float* A, float* B,
    int heightA, int widthA, int muly) {

    __shared__ float tile[TILE][TILE+1];

    int x = blockIdx.x * TILE + (threadIdx.x / muly);
    int y = blockIdx.y * TILE * muly + threadIdx.y + (threadIdx.x %
         muly)*TILE;

    int ind = y * widthA + x;
```

```
    if( x < widthA && y < heightA ) {
        tile[threadIdx.y][threadIdx.x] = A[ind];
    }
    __syncthreads();

    x = blockIdx.y * TILE * muly + threadIdx.x + (threadIdx.y %
        muly)*TILE;
    y = blockIdx.x * TILE + (threadIdx.y / muly);

    ind = y * heightA + x;
    if( x < heightA && y < widthA ) {
        B[ind] = tile[threadIdx.x][threadIdx.y];
    }
}
```

The general approach to choose between the three transpose algorithms, is shown below

```
if (height == 1 || width == 1) {
    copy or do nothing
} if (height <= TILE_DIM/2 && width > TILE_DIM) {
    transpose_lowheight
} else if (width <= TILE_DIM/2 && height > TILE_DIM) {
    transpose_lowwidth
else {
    transpose_current
}
```

Appendix D

Runtimes for All Benchmarks

In this appendix I will show runtimes for all the benchmarks mentioned in chapter 7. However, we will start with the data for the detailed breakdown of the Backprop and K-means benchmarks, so it does not get lost within all the figures in the following pages.

D.1 Backprop Detailed Breakdown

Only uses commutative reductions. This break down is for when using the `medium` dataset.

D.1.1 Vanilla

```
Kernel map_kernel_6217                          executed      1 times, with average runtime:    231us and
      total runtime:    231us
Kernel fut_kernel_map_transpose_f32             executed      1 times, with average runtime:   1017us and
      total runtime:   1017us
Kernel map_kernel_6290                          executed      1 times, with average runtime:   1054us and
      total runtime:   1054us
Kernel scan1_kernel_6391                        executed      1 times, with average runtime:   4507us and
      total runtime:   4507us <<<
Kernel map_kernel_6721                          executed      1 times, with average runtime:   1540us and
      total runtime:   1540us
Kernel fut_kernel_map_transpose_lowwidth_f32    executed      0 times, with average runtime:      0us and
      total runtime:      0us
Kernel chunked_reduce_kernel_6630               executed      1 times, with average runtime:    211us and
      total runtime:    211us
Kernel map_kernel_6190                          executed      1 times, with average runtime:   8942us and
      total runtime:   8942us
Kernel fut_kernel_map_transpose_lowheight_f32   executed      0 times, with average runtime:      0us and
      total runtime:      0us
Kernel map_kernel_6701                          executed      1 times, with average runtime:    865us and
      total runtime:    865us
Kernel map_kernel_6774                          executed      1 times, with average runtime:     22us and
      total runtime:     22us
Kernel map_kernel_6738                          executed      1 times, with average runtime:    976us and
      total runtime:    976us
Kernel map_kernel_6232                          executed      1 times, with average runtime:     38us and
      total runtime:     38us
Kernel map_kernel_6304                          executed      1 times, with average runtime:     26us and
      total runtime:     26us
Kernel scan2_kernel_6441                        executed      1 times, with average runtime:     32us and
      total runtime:     32us <<<
Kernel reduce_kernel_6595                       executed      1 times, with average runtime:     19us and
      total runtime:     19us
Kernel map_kernel_6494                          executed      1 times, with average runtime:     24us and
      total runtime:     24us
Kernel map_kernel_6792                          executed      1 times, with average runtime:     40us and
      total runtime:     40us
Kernel map_kernel_6262                          executed      1 times, with average runtime:    770us and
      total runtime:    770us
```

```
Kernel chunked_reduce_kernel_6569             executed      1 times, with average runtime:     23us and
      total runtime:     23us
Kernel chunked_reduce_kernel_6517             executed      1 times, with average runtime:     24us and
      total runtime:     24us
Kernel reduce_kernel_6684                     executed      1 times, with average runtime:     20us and
      total runtime:     20us
Kernel reduce_kernel_6543                     executed      1 times, with average runtime:     19us and
      total runtime:     19us
Kernel map_kernel_6333                        executed      1 times, with average runtime:    564us and
      total runtime:    564us
Kernel map_kernel_6480                        executed      1 times, with average runtime:   1276us and
      total runtime:   1276us
Ran 23 kernels with cumulative runtime:  22240us
```

Total time spent on computing segmented reductions: $32 + 4507 = 4539$, which is 20.41% of the total runtime.

D.1.2 Thesis (with segmented reduction)

```
Kernel map_kernel_6217                                        executed      1 times, with average runtime:
        203us and total runtime:    203us
Kernel fut_kernel_map_transpose_f32                           executed      1 times, with average runtime:
        995us and total runtime:    995us
Kernel segmented_redomap__one_group_one_segment_kernel_6467   executed      0 times, with average runtime:
          0us and total runtime:      0us <<<
Kernel chunked_reduce_kernel_6838                             executed      1 times, with average runtime:
        204us and total runtime:    204us
Kernel map_kernel_6290                                        executed      1 times, with average runtime:
       1020us and total runtime:   1020us
Kernel fut_kernel_map_transpose_lowwidth_f32                  executed      0 times, with average runtime:
          0us and total runtime:      0us
Kernel map_kernel_6909                                        executed      1 times, with average runtime:
        886us and total runtime:    886us
Kernel map_kernel_6946                                        executed      1 times, with average runtime:
        994us and total runtime:    994us
Kernel map_kernel_6190                                        executed      1 times, with average runtime:
       8946us and total runtime:   8946us
Kernel map_kernel_6929                                        executed      1 times, with average runtime:
       1530us and total runtime:   1530us
Kernel fut_kernel_map_transpose_lowheight_f32                 executed      0 times, with average runtime:
          0us and total runtime:      0us
Kernel segmented_redomap__one_group_one_segment_kernel_6359   executed      0 times, with average runtime:
          0us and total runtime:      0us <<<
Kernel map_kernel_6232                                        executed      1 times, with average runtime:
         38us and total runtime:     38us
Kernel map_kernel_6304                                        executed      1 times, with average runtime:
         27us and total runtime:     27us
Kernel reduce_kernel_6751                                     executed      1 times, with average runtime:
         20us and total runtime:     20us
Kernel reduce_kernel_6892                                     executed      1 times, with average runtime:
         23us and total runtime:     23us
Kernel chunked_reduce_kernel_6777                             executed      1 times, with average runtime:
         24us and total runtime:     24us
Kernel chunked_reduce_kernel_6725                             executed      1 times, with average runtime:
         26us and total runtime:     26us
Kernel segmented_redomap__one_group_many_segment_kernel_6650  executed      0 times, with average runtime:
          0us and total runtime:      0us <<<
Kernel map_kernel_6982                                        executed      1 times, with average runtime:
         22us and total runtime:     22us
Kernel map_kernel_7000                                        executed      1 times, with average runtime:
         41us and total runtime:     41us
Kernel map_kernel_6262                                        executed      1 times, with average runtime:
        789us and total runtime:    789us
Kernel segmented_redomap__one_group_many_segment_kernel_6497  executed      1 times, with average runtime:
         25us and total runtime:     25us <<<
Kernel segmented_redomap__many_groups_one_segment_kernel_6417 executed      1 times, with average runtime:
        338us and total runtime:    338us <<<
Kernel reduce_kernel_6803                                     executed      1 times, with average runtime:
         19us and total runtime:     19us
Ran 20 kernels with cumulative runtime:  16170us
```

Using the large kernel with multiple groups per segment, and one group per segment to reduce this recursively.

Total time spent on computing segmented reductions: $25 + 338 = 363$ which is 2.24% of the total runtime.

D.2 K-Means Detailed Breakdown

Only uses commutative reductions. This break down is for when using the `kdd_cup` dataset.

D.2.1 Vanilla

```
Kernel map_kernel_2370                          executed      1 times, with average runtime:    211us and
      total runtime:    211us
Kernel reduce_kernel_2904                       executed     37 times, with average runtime:     15us and
      total runtime:    560us
Kernel fut_kernel_map_transpose_f32             executed     38 times, with average runtime:    303us and
      total runtime:  11547us
Kernel scan2_kernel_2795                        executed     37 times, with average runtime:     20us and
      total runtime:    753us <<<
Kernel fut_kernel_map_transpose_i32             executed     74 times, with average runtime:     40us and
      total runtime:   3006us
Kernel kernel_copy_3313                         executed      1 times, with average runtime:    818us and
      total runtime:    818us
Kernel fut_kernel_map_transpose_lowwidth_f32    executed      0 times, with average runtime:      0us and
      total runtime:      0us
Kernel chunked_map_kernel_2415                  executed     37 times, with average runtime:   7372us and
      total runtime: 272796us
Kernel fut_kernel_map_transpose_lowheight_f32 executed      0 times, with average runtime:      0us and
      total runtime:      0us
Kernel kernel_copy_3319                         executed      1 times, with average runtime:    209us and
      total runtime:    209us
Kernel kernel_copy_3386                         executed     37 times, with average runtime:     14us and
      total runtime:    550us
Kernel scan1_kernel_2500                        executed     37 times, with average runtime:     58us and
      total runtime:   2160us <<<
Kernel map_kernel_2603                          executed     37 times, with average runtime:     14us and
      total runtime:    525us
Kernel map_kernel_2851                          executed     37 times, with average runtime:     15us and
      total runtime:    589us
Kernel map_kernel_2675                          executed      1 times, with average runtime:    254us and
      total runtime:    254us
Kernel kernel_copy_3316                         executed      1 times, with average runtime:    243us and
      total runtime:    243us
Kernel fut_kernel_map_transpose_lowheight_i32 executed     37 times, with average runtime:     36us and
      total runtime:   1348us
Kernel map_kernel_2834                          executed     37 times, with average runtime:    287us and
      total runtime:  10644us
Kernel chunked_map_kernel_2642                  executed     37 times, with average runtime:   2558us and
      total runtime:  94665us
Kernel map_kernel_2589                          executed     37 times, with average runtime:     27us and
      total runtime:   1031us
Kernel kernel_copy_3383                         executed     37 times, with average runtime:     85us and
      total runtime:   3160us
Kernel chunked_reduce_kernel_2873               executed     37 times, with average runtime:     57us and
      total runtime:   2131us
Kernel map_kernel_2442                          executed      1 times, with average runtime:    197us and
      total runtime:    197us
Kernel scan1_kernel_2745                        executed     37 times, with average runtime:   2843us and
      total runtime: 105214us <<<
Kernel scan2_kernel_2550                        executed     37 times, with average runtime:     17us and
      total runtime:    635us <<<
Kernel fut_kernel_map_transpose_lowwidth_i32  executed      0 times, with average runtime:      0us and
      total runtime:      0us
Ran 673 kernels with cumulative runtime: 513246us
```

Total time spent on computing segmented reductions: $753 + 2160 + 105214 + 635 = 108762$ microseconds, which is 21.19% of the total runtime.

D.2.2 Thesis (with segmented reduction)

```
Kernel map_kernel_2370                                          executed      1 times, with average runtime:
        366us and total runtime:    366us
Kernel segmented_redomap_many_groups_one_segment_kernel_2968 executed      0 times, with average runtime:
          0us and total runtime:      0us
Kernel fut_kernel_map_transpose_f32                             executed      1 times, with average runtime:
       1183us and total runtime:   1183us
Kernel segmented_redomap_one_group_many_segment_kernel_2606  executed     37 times, with average runtime:
         17us and total runtime:    649us <<<
Kernel kernel_copy_3712                                         executed      1 times, with average runtime:
        283us and total runtime:    283us
Kernel kernel_copy_3785                                         executed     37 times, with average runtime:
         86us and total runtime:   3208us
Kernel fut_kernel_map_transpose_i32                             executed     74 times, with average runtime:
         40us and total runtime:   3029us
```

```
Kernel fut_kernel_map_transpose_lowwidth_f32                  executed      0 times, with average runtime:
           0us and total runtime:      0us
Kernel segmented_redomap__many_groups_one_segment_kernel_2526 executed     37 times, with average runtime:
          26us and total runtime:    993us <<<
Kernel kernel_copy_3788                                       executed     37 times, with average runtime:
          15us and total runtime:    565us
Kernel segmented_redomap__one_group_many_segment_kernel_3207  executed      0 times, with average runtime:
           0us and total runtime:      0us <<<
Kernel segmented_redomap__one_group_one_segment_kernel_2909   executed     37 times, with average runtime:
         109us and total runtime:   4045us <<<
Kernel chunked_map_kernel_2415                                executed     37 times, with average runtime:
        7753us and total runtime: 286869us
Kernel segmented_redomap__one_group_many_segment_kernel_3049  executed      0 times, with average runtime:
           0us and total runtime:      0us <<<
Kernel fut_kernel_map_transpose_lowheight_f32                 executed      0 times, with average runtime:
           0us and total runtime:      0us
Kernel kernel_copy_3715                                       executed      1 times, with average runtime:
         203us and total runtime:    203us
Kernel chunked_map_kernel_2850                                executed     37 times, with average runtime:
        2575us and total runtime:  95298us
Kernel segmented_redomap__one_group_one_segment_kernel_2576   executed      0 times, with average runtime:
           0us and total runtime:      0us <<<
Kernel chunked_reduce_kernel_3282                             executed     37 times, with average runtime:
          61us and total runtime:   2288us
Kernel segmented_redomap__one_group_one_segment_kernel_3019   executed      0 times, with average runtime:
           0us and total runtime:      0us <<<
Kernel segmented_redomap__one_group_one_segment_kernel_2468   executed      0 times, with average runtime:
           0us and total runtime:      0us <<<
Kernel fut_kernel_map_transpose_lowheight_i32                 executed      0 times, with average runtime:
           0us and total runtime:      0us
Kernel segmented_redomap__one_group_many_segment_kernel_2759  executed      0 times, with average runtime:
           0us and total runtime:      0us <<<
Kernel reduce_kernel_3313                                     executed     37 times, with average runtime:
          16us and total runtime:    627us
Kernel kernel_copy_3709                                       executed      1 times, with average runtime:
         881us and total runtime:    881us
Kernel fut_kernel_map_transpose_lowwidth_i32                  executed      0 times, with average runtime:
           0us and total runtime:      0us
Ran 412 kernels with cumulative runtime: 400487us
```

Usaing the large kernel with multiple groups per segment, one group per segment, and the small kernel.

Total time spent on computing segmented reductions: $993 + 4045 + 649 = 5687$ microseconds, which is 1.1% of the total runtime.

D.3 Speedups

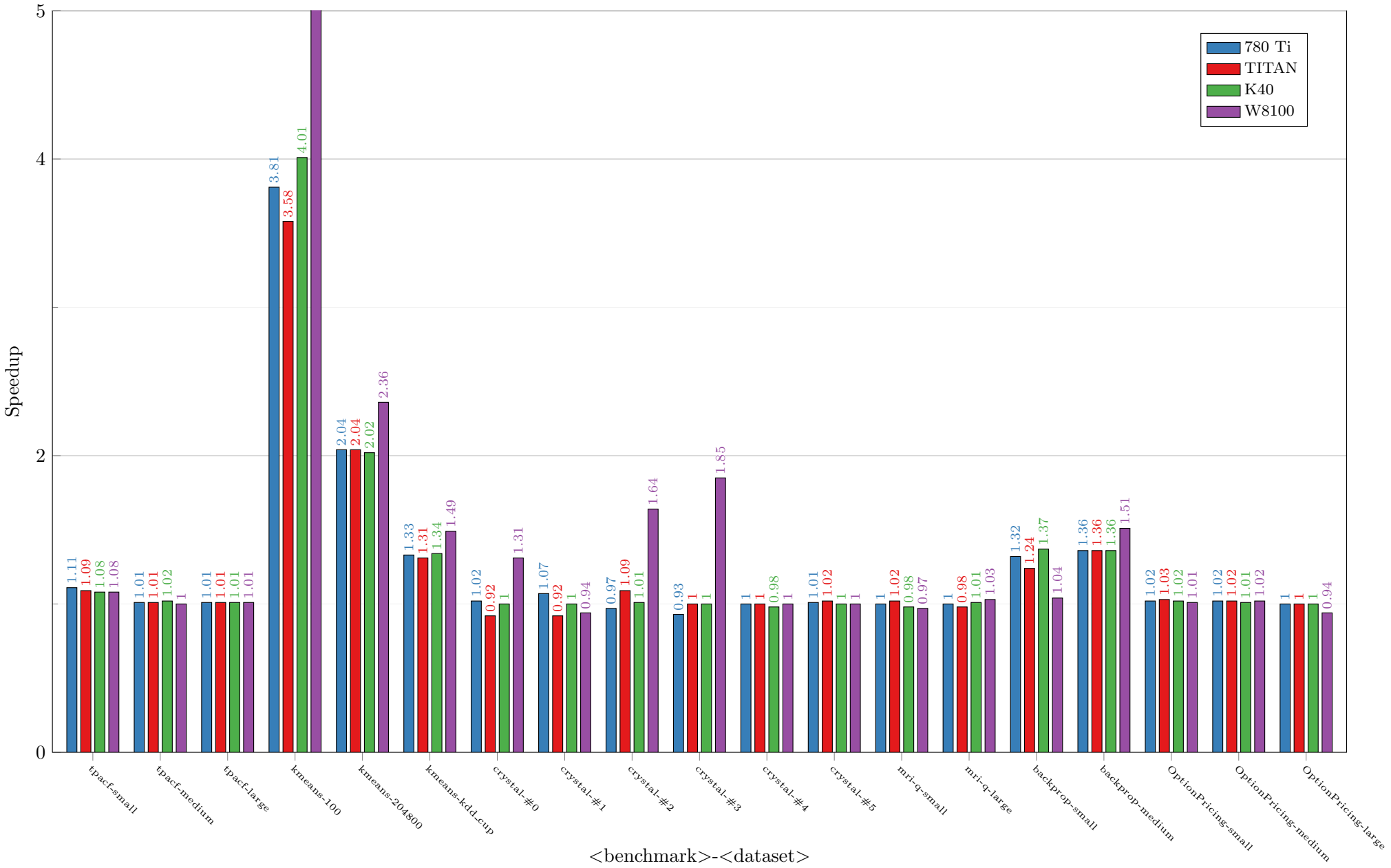


Figure D.1: Speedups achieved by using my implementation for segmented reductions over vanilla Futhark, on four different GPUs. All benchmarks that cotained either a segmented reduction or a segmented redomap is shown.

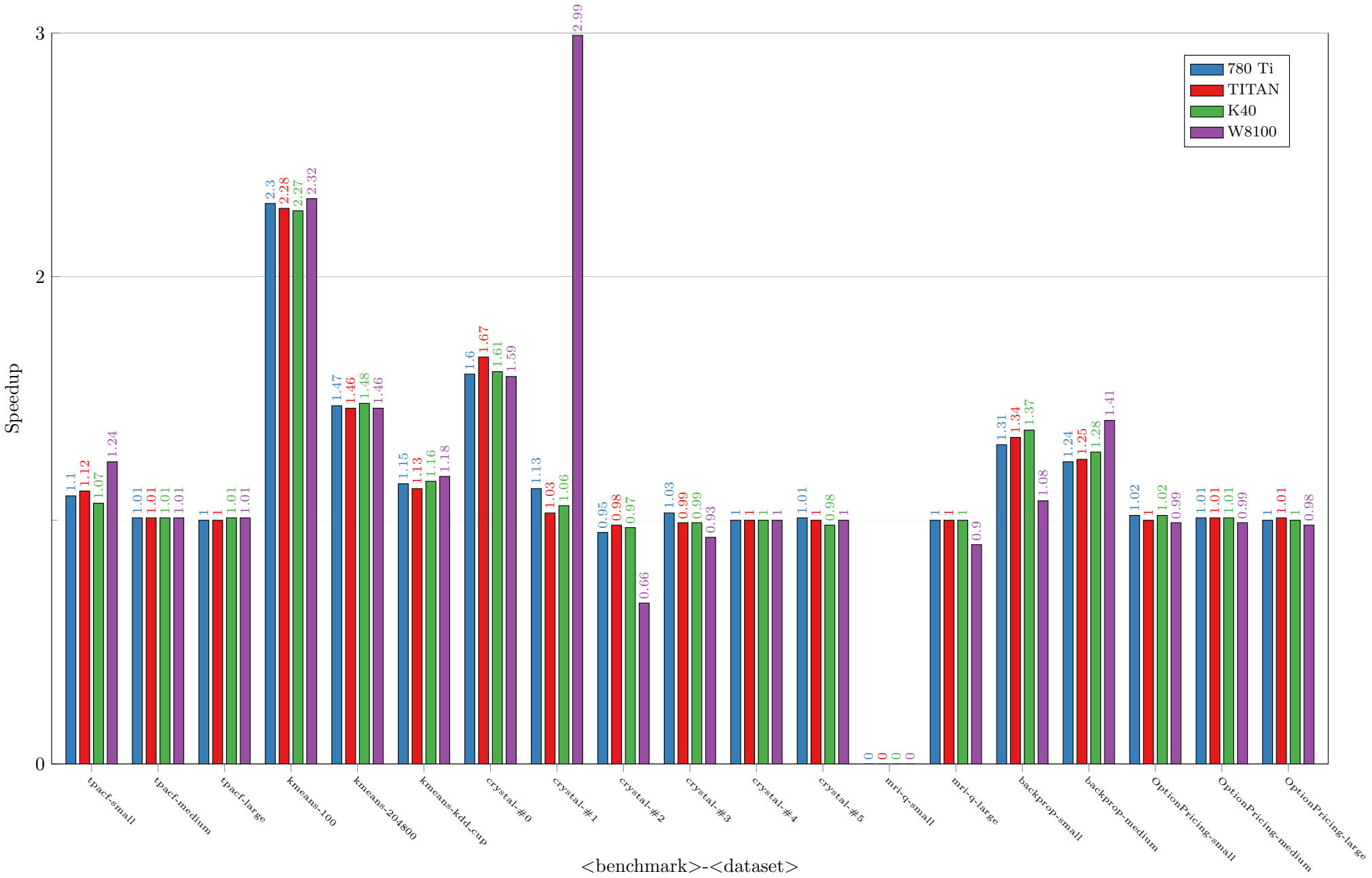
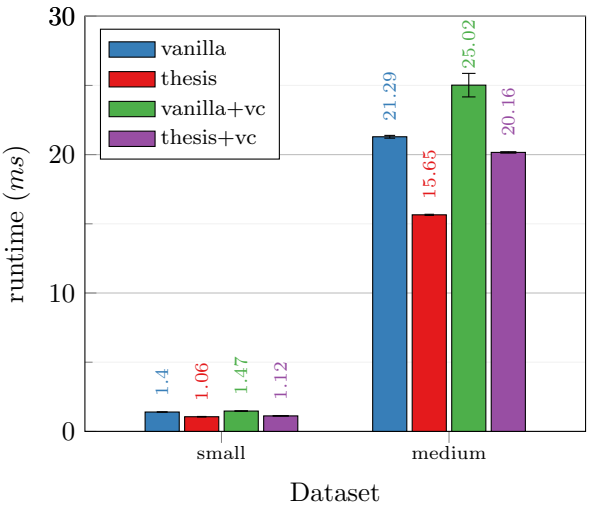
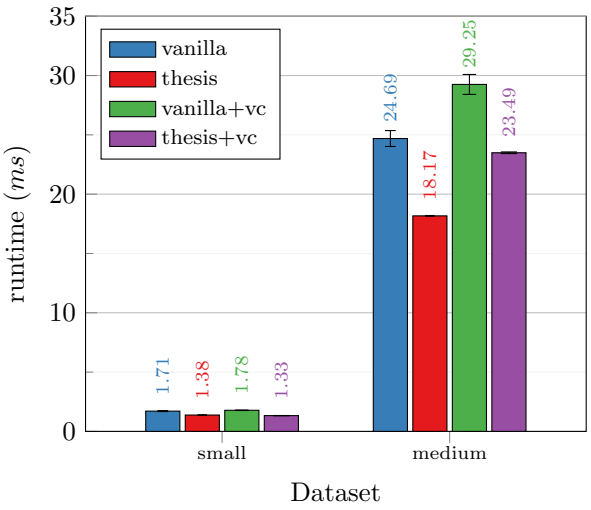


Figure D.2: Speedups achieved by using versioned code and my implementation for segmented reduction and segmented redomaps over Futhark with versioned code, on four different GPUs.
All benchmarks that cotained either a segmented reduction or a segmented redomap is shown.

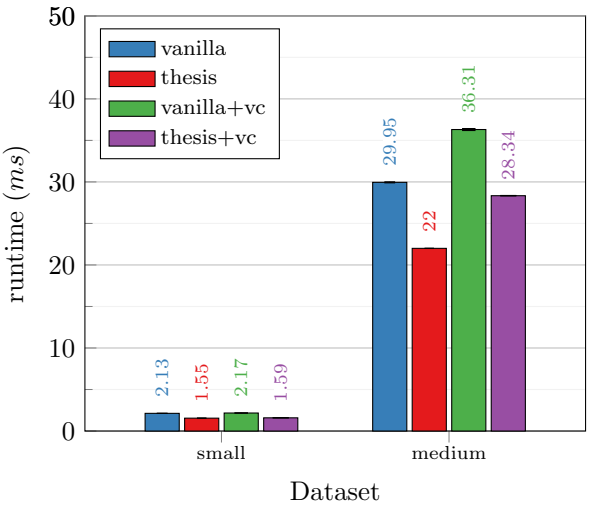
D.4 Runtimes



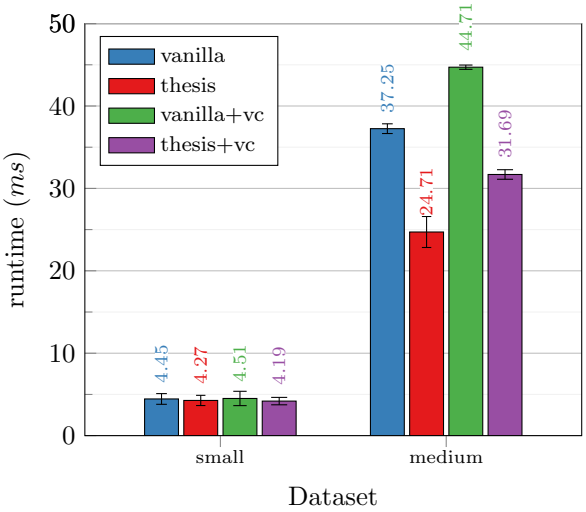
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU

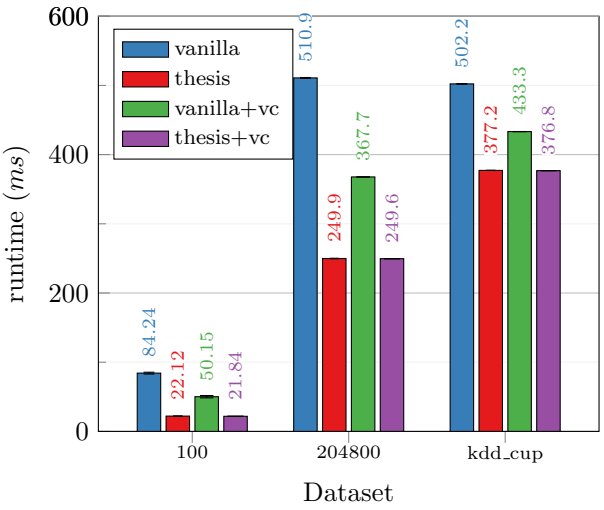


(c) Runtime on the NVIDIA Tesla K40 GPU

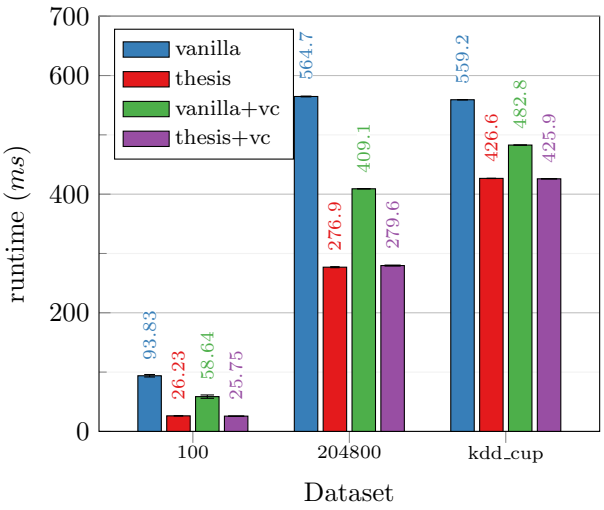


(d) Runtime on the AMD FireGL W8100

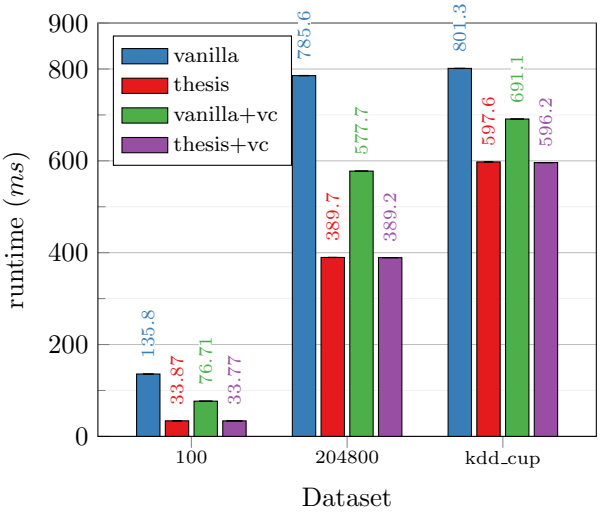
Figure D.3: Runtimes for the Backprop benchmark from Rodinia. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.



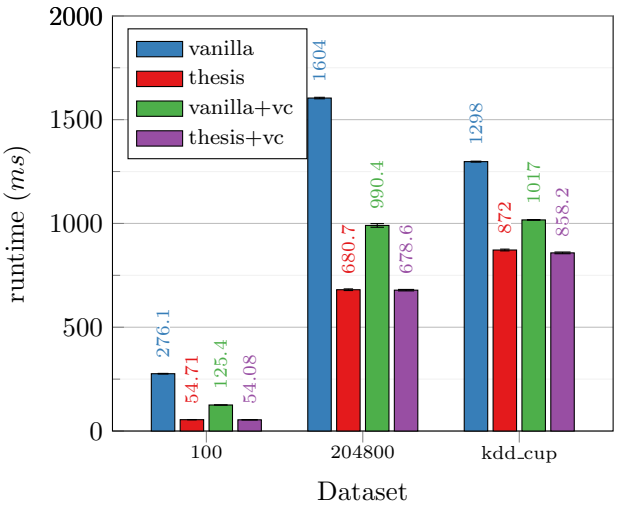
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU

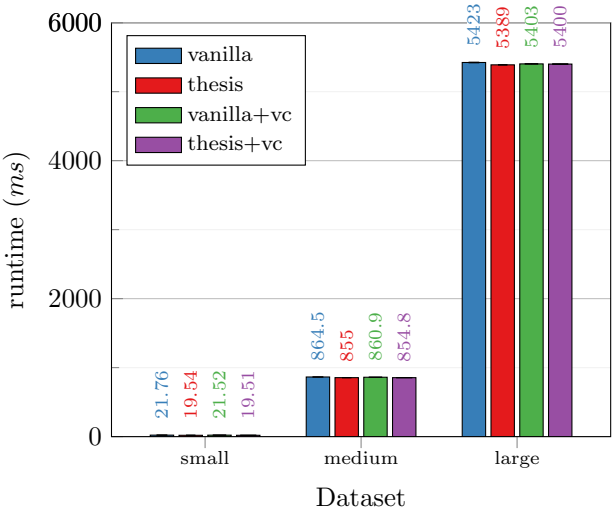


(c) Runtime on the NVIDIA Tesla K40 GPU

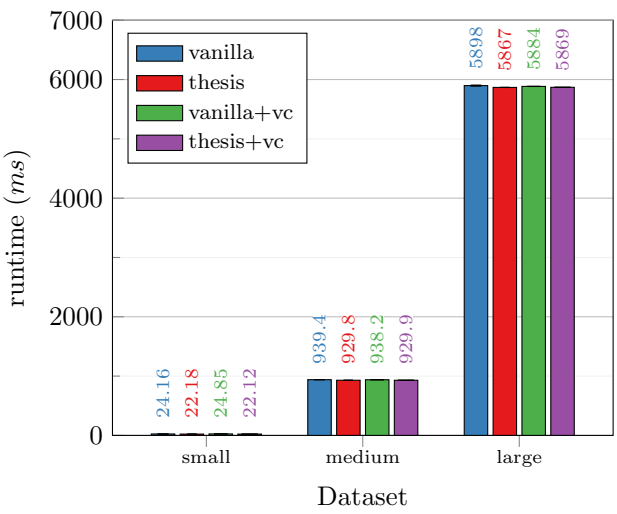


(d) Runtime on the AMD FireGL W8100

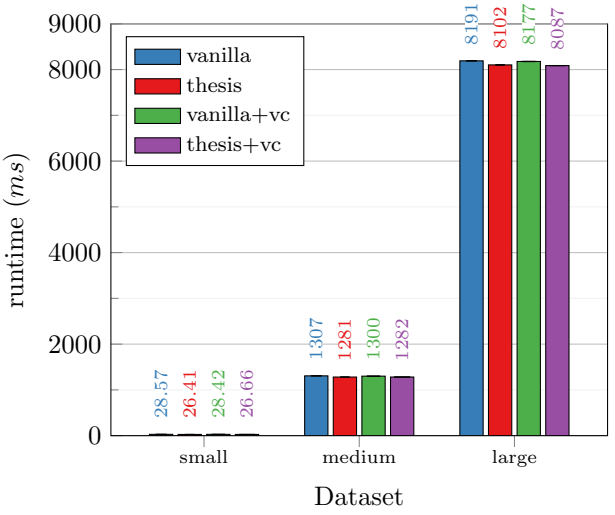
Figure D.4: Runtimes for the K-means benchmark from Rodinia. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.



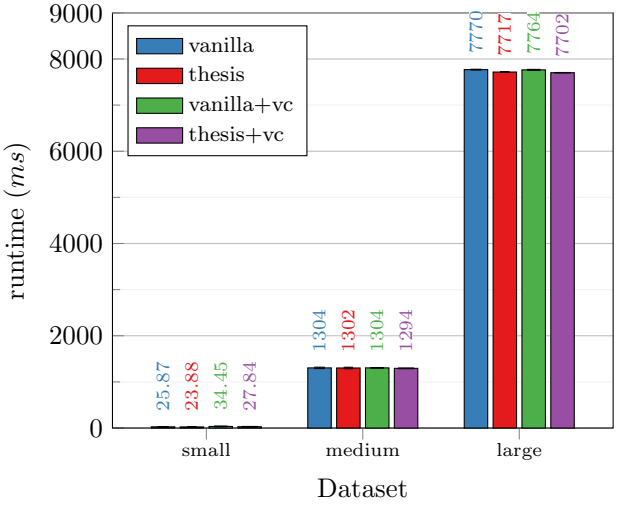
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU

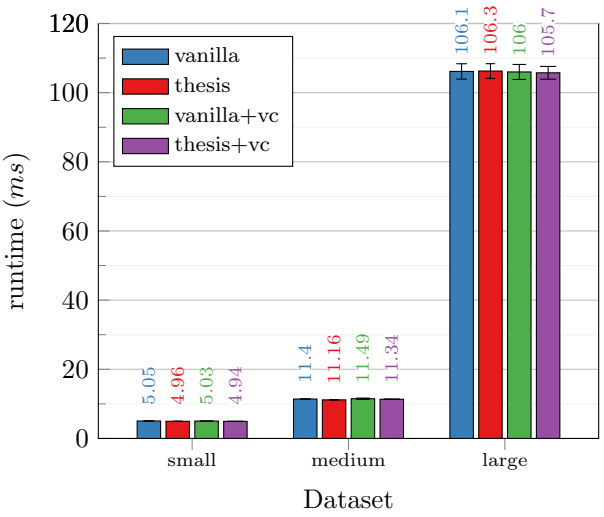


(c) Runtime on the NVIDIA Tesla K40 GPU

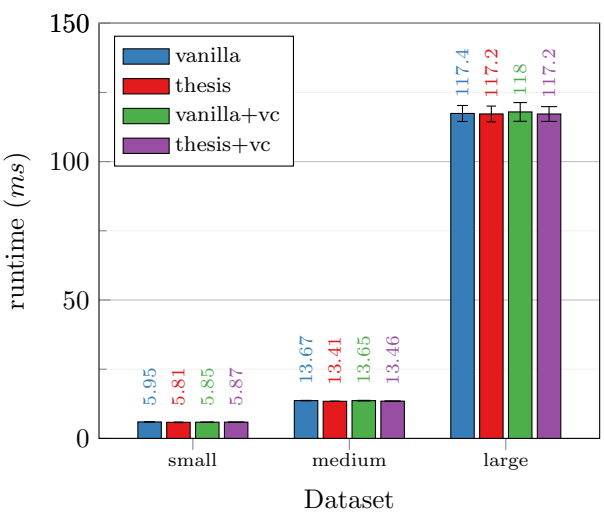


(d) Runtime on the AMD FireGL W8100

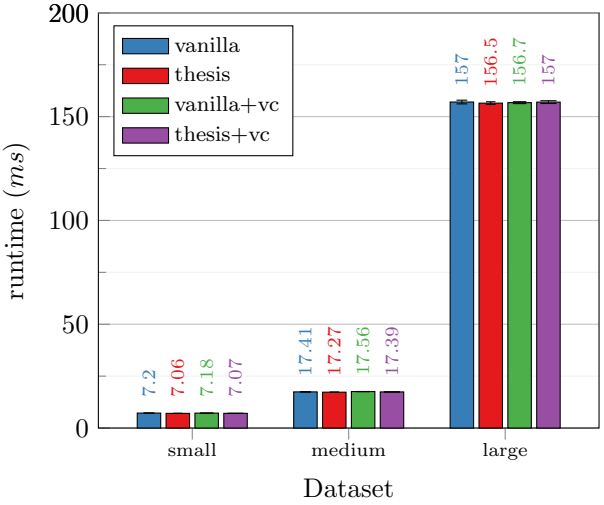
Figure D.5: Runtimes for the TPACF benchmark from Parboil. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.



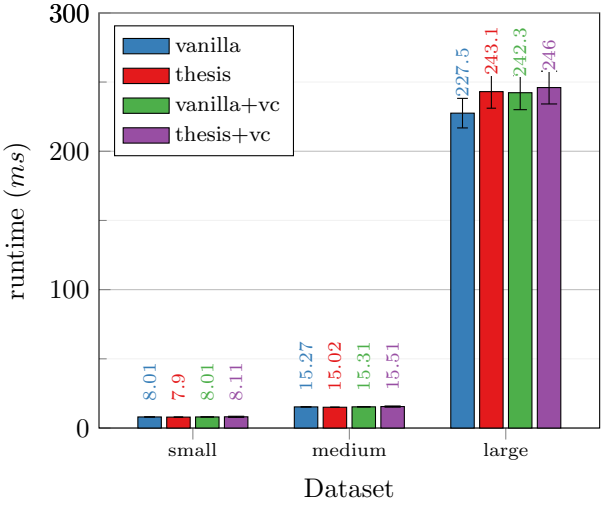
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU

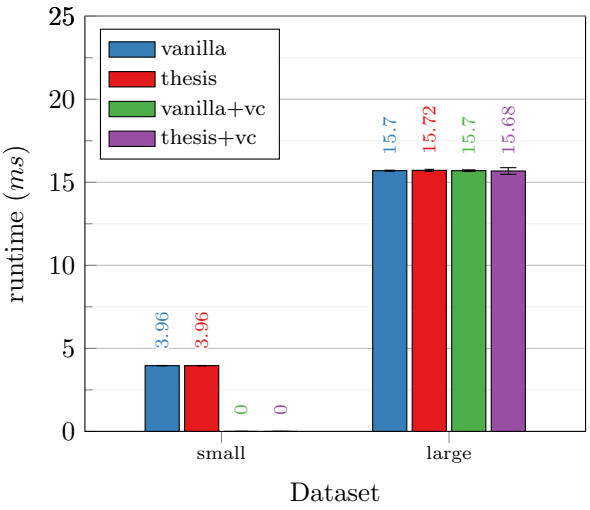


(c) Runtime on the NVIDIA Tesla K40 GPU

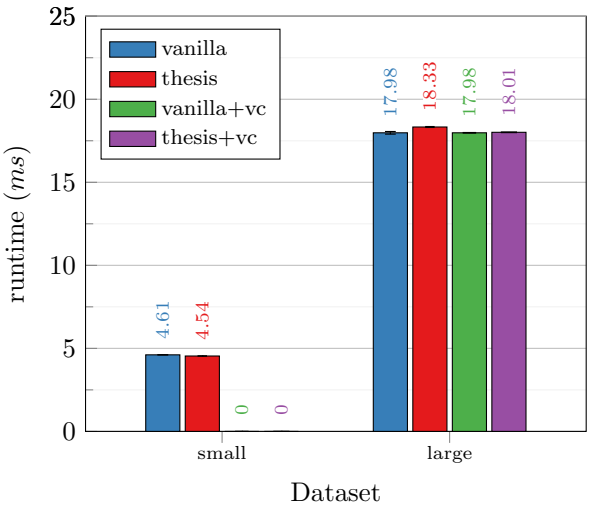


(d) Runtime on the AMD FireGL W8100

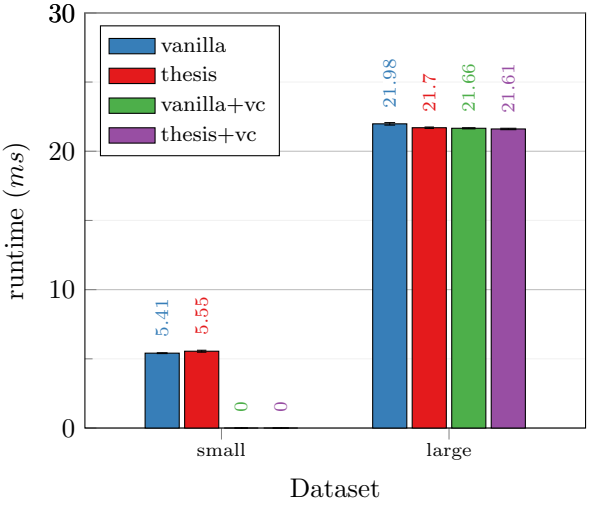
Figure D.6: Runtimes for the OptionPricing benchmark from FinPar. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.



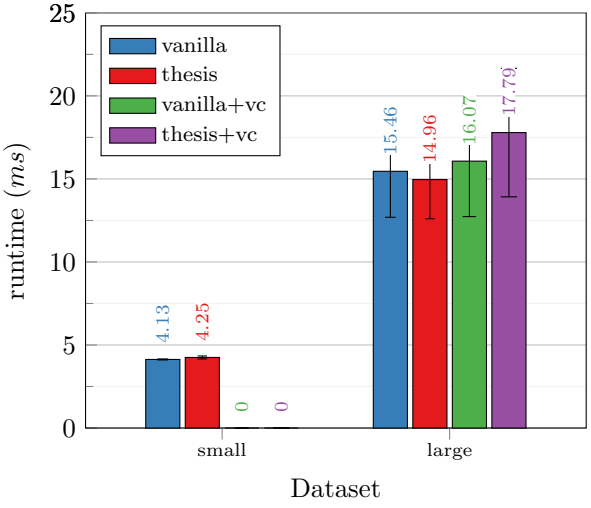
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU

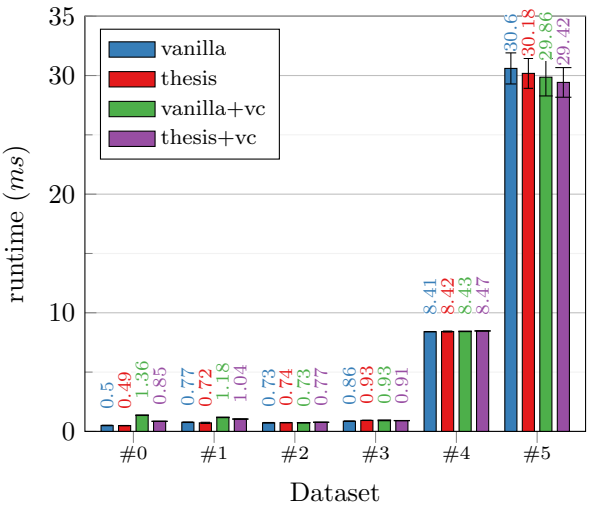


(c) Runtime on the NVIDIA Tesla K40 GPU

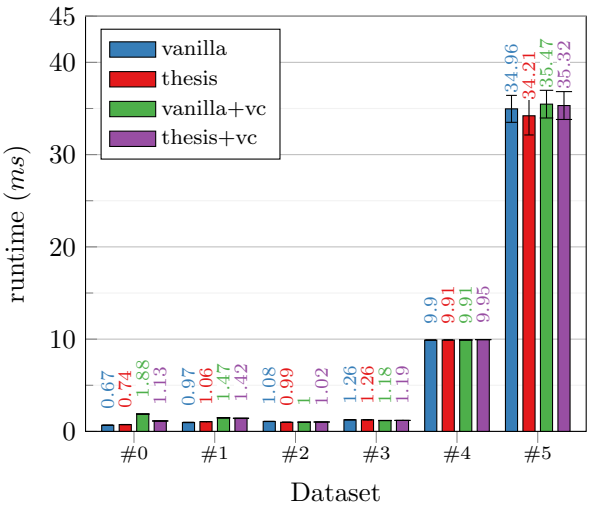


(d) Runtime on the AMD FireGL W8100

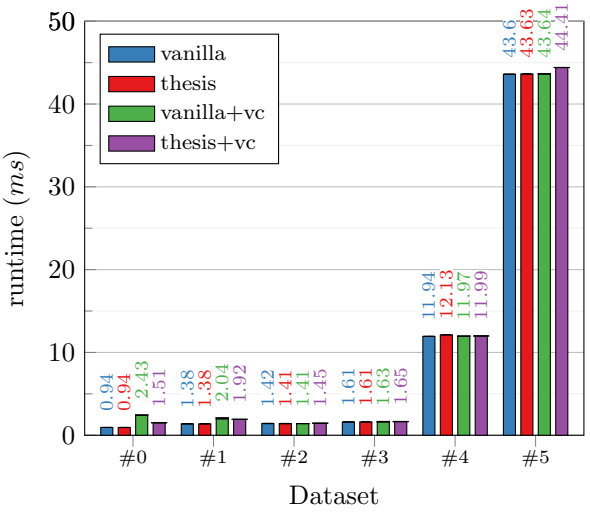
Figure D.7: Runtimes for the MRI-Q benchmark from Parboil. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.



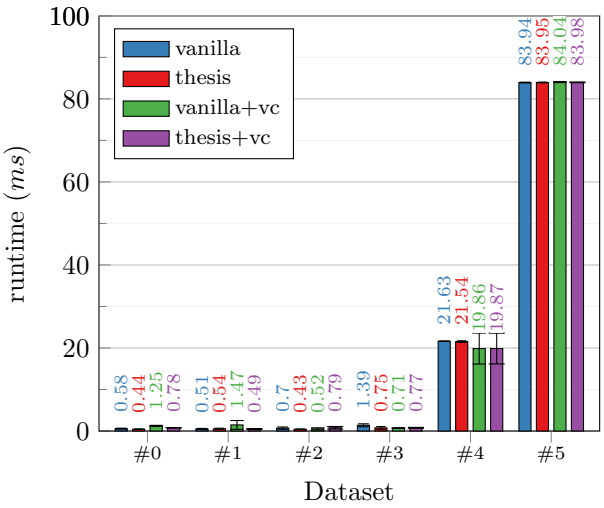
(a) Runtime on the NVIDIA GTX 780 Ti GPU



(b) Runtime on the NVIDIA GTX TITAN Black GPU



(c) Runtime on the NVIDIA Tesla K40 GPU



(d) Runtime on the AMD FireGL W8100

Figure D.8: Runtimes for the Crystal benchmark from Accelerate. *vanilla* is the Futhark compiler without modifications, *thesis* is the Futhark compiler with my implementation for segmented reductions. The *+vc* versions uses versioned code to, at runtime, select between using an implementation using the loop-in-map strategy, or the implementation for a segmented reduction.