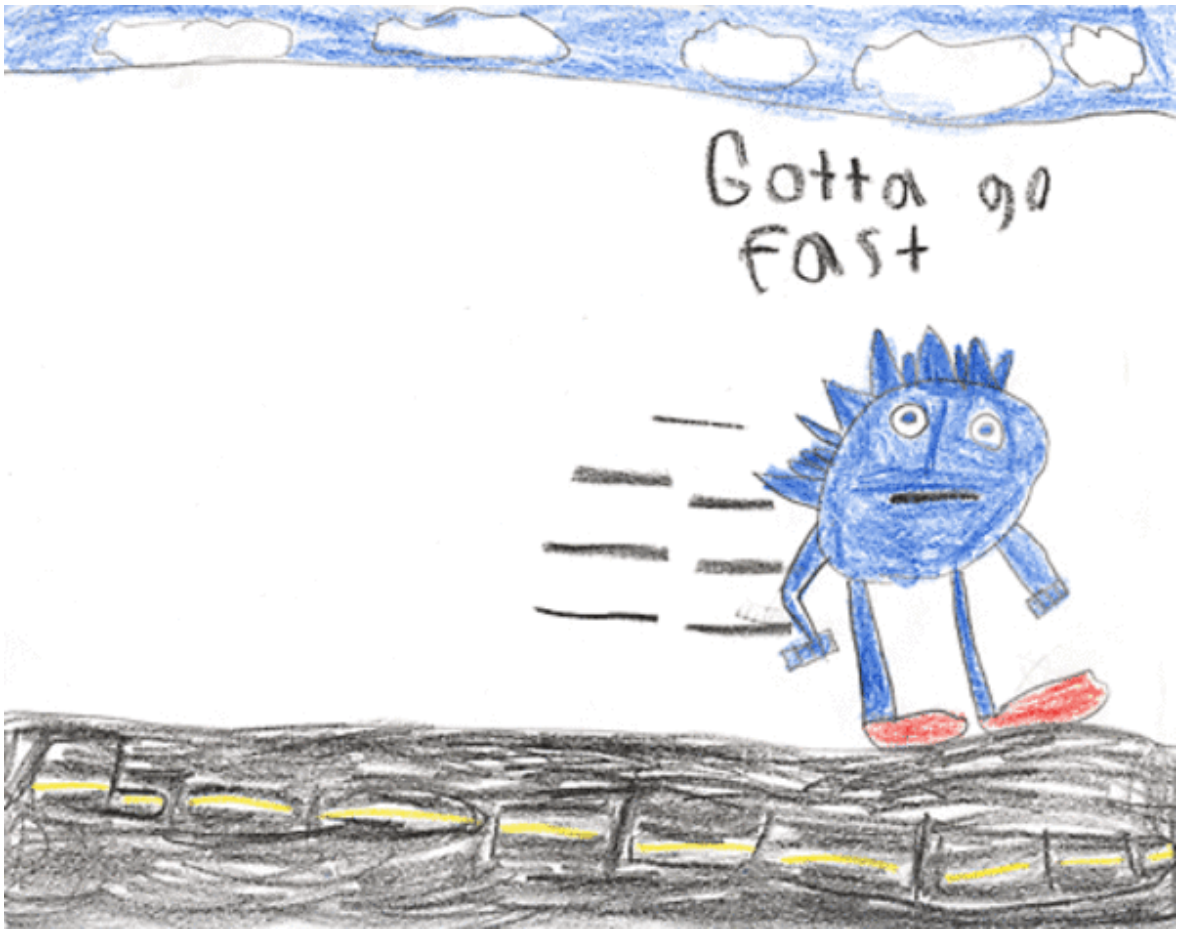
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Master's Thesis

Sune Hellfritzsch

Efficient Histogram Computation on GPGPUs

Supervisors: Cosmin Eugen Oancea and Troels Henriksen
Submitted: October 23, 2018

Abstract

In this thesis, we describe the development and implementation of a new language construct for efficient computation of generalised reductions in the programming language Futhark. A generalised reduction, which also goes by the names *reduction by key* or *reduction by index*, is reducing a collection of data into k buckets where there is no pattern in the input data. This is opposed to segmented reductions where input data is grouped by segments. Throughout the thesis, we use the computation of a traditional histogram – a concrete instance of generalised reductions – as a running example.

We show how such a random reduction pattern can effectively be implemented on graphics processing units (GPUs) by using atomic functions. The implementation is based on the idea of letting GPU threads cooperate on subhistograms, i.e., the input data is split between groups of threads such that each group produce their own local histogram, which are ultimately combined into one final histogram. For choosing a number of cooperating threads we present a simple heuristic, which are based on a comprehensive experiment also presented in the thesis.

Finally, we evaluate the new construct on a collection of adversarial datasets and we show that it performs at least as well, and often much better, than existing solutions in Futhark.

Contents

1 Introduction

This thesis describes the design, development, and implementation of a new language construct in the programming language Futhark and its optimizing compiler. The construct provides a way to efficiently compute what is referred to as *generalised reductions*. Before we dive into details let us gain some intuition for the new construct by an example, and at the same time explore at a very high level some of the problems we need to address.

The example is a histogram computation which is a well-known instance of a general reduction. Say you have an image and want to count the frequency of each color. The simplest algorithm is to process one pixel at a time: determine its color and increment the corresponding counter by one until all pixels have been processed. The result can be visualized as a histogram as shown in Figure 1, and the algorithm is here given in C-like pseudo-code:

```
for(int i=0; i<N; i++) {
  col = f(image[i])
  cnt = hist[col]
  hist[col] = cnt + 1
}
```

where `image` is an array of size `N` containing a flattened version of the image, `hist` is an array whose length is equal to the number of unique colors in the

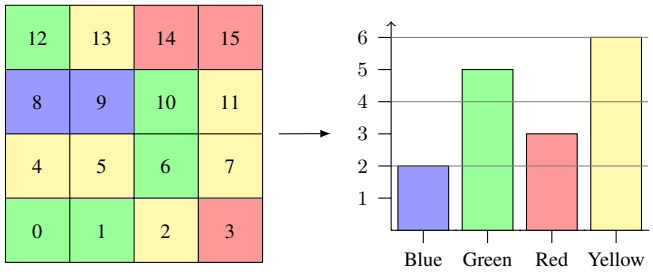


Figure 1: Histogram computation for an image. Each bucket has height corresponding the number of occurences of the color.

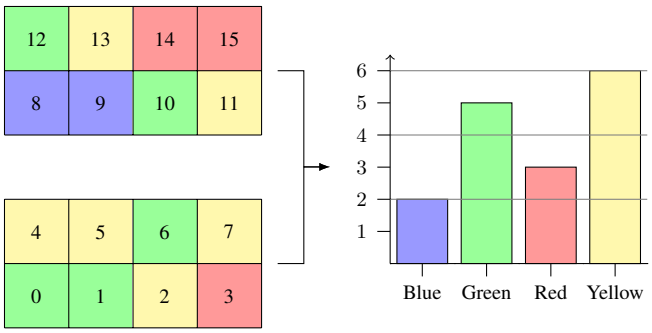


Figure 2: Data races. What happens when we simultaneously process two pixels that goes into the same bucket?

image, and `f` is a bucket function computing the index, called a bucket, into `hist` for any color in `image`.

In a sequential setting, this approach is very slow but also completely safe. One way to speed up the process is to realize that determining the color of one pixel is independent of determining the color of any other pixel. Thus, you could half the time taken to compute the histogram by *cooperating* with a friend by splitting the image in two equally-sized parts and have your friend process one half while you process the other half. This idea is displayed in Figure 2.

But this approach comes with a cost as it implicitly introduces *data races*. As an example, assume that you and your friend processes pixels 5 and 13 at the same time. Then both of you will read the current value of the counter for yellow, namely, 2, increment the value locally by one, to 4, and write back the value simultaneously. Because you read the same value the new value after both writes will be 4 but it should have been 5.

Such data races can be avoided by using *atomic operations*. This means that one can perform a read-modify-write operation without being interfered by someone else trying to read or write the same piece of data. Effectively, this serializes simultaneous accesses to the same piece of data as you or your friend must wait for the other to finish her computations. We will go into detail about how such atomic operations can be used but for now, we assume that they exist and that we can use them.

The observant reader may notice that in the worst case, where all pixels have the same color, this naive application of atomic operations corresponds to the sequential version described in pseudo-code above. Thus, we are also concerned with the level of *memory contention*, i.e., the number of accesses to the same bucket in the same histogram. Continuing the example, we would like to minimize contention between you and your friend in order to avoid expensive serialization.

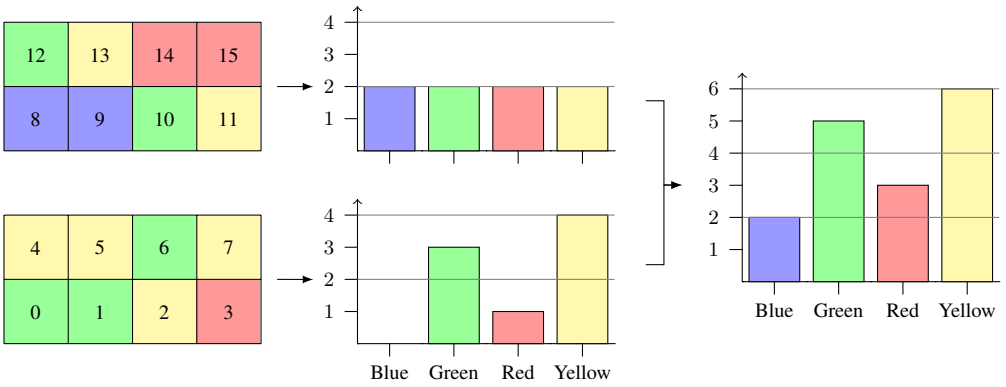


Figure 3: Memory contention. We split the image in half and compute two local histograms before combining them into a final histogram.

One such method for minimizing contention we will call *subhistogramming* which is shown in Figure 3. The idea is, that instead of cooperating on the same histogram you compute your own local histogram. When all local histograms are computed one person combines them into the final histogram. In the worst case described above this approach will half the time needed to compute the histogram not counting in the time needed to combine the subhistograms. It turns out that this idea scales rather elegantly if you are careful with certain subtle issues. We will return to these issues later.

Before the example, we claimed that a histogram is a specific instance of a pattern called generalized reductions. But that was vague because we did not explain what generalized reductions are, and it is a fair question to ask how histograms and generalized reductions relate to each other. Below we have rewritten the histogram computation from above (left) and shown how it corresponds to a generalised reduction pattern (right):

```
for(int i=0; i<N; i++) {
  col, _ = f(image[i])
  cnt = hist[col]
  new = cnt + 1
  hist[col] = new
}
```

$\Rightarrow$

```
for(int i=0; i<N; i++) {
  ind, val = f(xs[i])
  old = ys[ind]
  new = old `op` val
  ys[ind] = new
}
```

where `xs` is an input array of size `N`, `ys` is the result array, `f` is a bucket function computing an index and a value, and `op` is an assocaitive and commutative binary operator. In particular it uses the operator, `op`, to combine `old` and `val` into a new value, `new`, which is stored at the same index, `ind`. If multiple `x[i]`'s produce the same index, `ind`, the corresponding values will be combined using

the original value in `ys[ind]` as the base value, hence "reducing" the input array into k buckets. This is opposed to the probably better known `reduce` operator in many functional languages, that reduces a collection of values into one value, which is effectively one bucket, by using a combining operator and a neutral element.

The running example above presented the main ideas in this thesis without mentioning the specific architecture, although most readers probably thought of graphics processing units (GPUs). This thesis investigates the aspects of general reductions presented in the above example in a parallel perspective, both theoretically and practically:

- Our main contribution is the design and implementation of a general reduction construct in the programming language Futhark and its optimizing compiler (Chapter 5). The implementation is capable of generating efficient GPU code based on user-provided input, i.e., at compile-time it selects the optimal strategy for implementing a combining function provided by the user, and at run-time it uses subhistogramming to mitigate the impact on runtime performance caused by the serializing effect from collisions.

- Since the primary code generation target is efficient GPU code, we establish a necessary background to understand the implementation (Chapter 2). In particular, we explain the programming model proposed by CUDA, along with well-known strategies for obtaining good performance on GPUs. Furthermore, since we implement the construct in the programming language Futhark, we explain the programming model of Futhark. We also explain some of its language features that are used throughout this thesis.

- In Chapter 3 we analyze the strengths and weaknesses of high-level strategies for computing histograms, in terms of their work complexities. The chapter also presents a small selection of related literature and work on histogram computations and generalised reductions. Finally, it discusses current solutions for computing histograms in Futhark, which serves as justification for our new construct.

- Chapter 4 builds the skeleton on which the code generation in the compiler implementation is based. Specifically, we 1) present three different code generations for implementing atomic operations provided by the user, and, since atomic functions serializes accesses on collision, we also 2) present the idea of subhistogramming, i.e., the number of cooperating threads per subhistogram. Finally, we run a comprehensive experiment investigating

the impact of cooperation level on runtime performance, and based on the results we propose a heuristic for choosing a cooperation level.

- The runtime performance of the new construct, using an optimized case for addition is compared to a sequential implementation of a traditional histogram, along with existing solutions in Futhark, and a single reference implementation in Thrust (Chapter 6), on 12 adversarial datasets. On all datasets we report speedups compared to the sequential solution, ranging from the smallest speedup of $1.62\times$ up to the largest speedup of $17.63\times$. On all datasets we greatly outperform both the sequential solution, existing solutions in Futhark, and the Thrust implementation (except for one case, where Futhark was already known to perform poorly).

  In addition, we rewrite two existing Futhark benchmarks to use the new construct. Here we see both a slowdown and a small speedup; both are known cases of where existing solutions perform really well.

- As mentioned in the previous bullet, the implemented construct is known to have some weaknesses, i.e., for large histograms where only a few buckets are hit it performs poorly. In Chapter 7 we briefly discuss if the limitations can be worked around, and based on the latter weakness we suggest as future study to look into caching behavior with respect to random writes

Part I

Realm of the Problem

2 Background

This chapter lays the foundation on which rest of the thesis is built. First, we describe the CUDA programming model which the final code generation is based on, along with a brief conceptual introduction to GPU accelerated programming. Second, we describe the design of a GPU – a concrete instance of a parallel architecture. Third, because current languages for writing parallel programs for GPUs, e.g., CUDA, are often highly sensitive to the given architecture, we discuss how to obtain good performance on GPUs. Finally, we introduce the programming language Futhark.

2.1 The CUDA Programming Model

To use NVIDIAs own words, CUDA is *a general purpose parallel computing platform and programming model*.

The platform part is a software layer including language extensions, compiler, debugger, drivers, and runtime environment. In this thesis, we will use the CUDA language extension to C/C++, but other languages and directive-based approaches are supported, e.g., Fortran and OpenACC. We use the platform part as is, and will not treat it further.

The programming model part is often overlooked because the language extensions implementing the programming model put a lot of focus on hardware. Nevertheless, we will use the C/C++ language extension as a vehicle for understanding parts of the programming model relevant to this thesis. But first, we will give a brief, conceptual introduction to how GPU accelerated programming works.

2.1.1 Heterogeneous programming

The CUDA programming model is a heterogeneous model which refers to the fact that more than one kind of processor is used in order to obtain performance gains. In our setting, we have that a CPU, called the *host*, is offloading heavy computations to a GPU, called the *device*. At this point, it suffices to know that

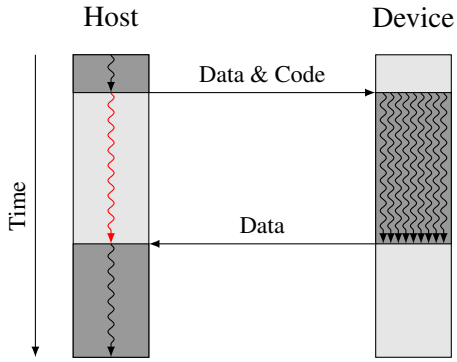


Figure 4: Heterogeneous programming. The host is using the GPU as a co-processor to accelerate computations. Note, that the host is always in control and may terminate the program running at the device at any time; the host may wait or continue its own computations.

a GPU is a physically separate device from the CPU that is able to efficiently execute multithreaded code. We will return to the architecture later.

Usually, the structure of a host program is as follows (see also Figure 4):

1. Declare and allocate host and device memory.
2. Initialize host memory.
3. Copy data from host to device.
4. Execute program on device.
5. Copy data from device to host.

Importantly, as also indicated by the figure, that the host is always in control, i.e., the host may terminate the program running on the device at any time.

During its lifetime the host can launch multiple programs running on the device, and the programs may also be able to run concurrently on the device depending on its capability.

2.1.2 Kernels

Programs that are to execute on a device are called kernels, not to be confused with operating system kernels. From the programmers point of view kernels can

be seen as regular functions in C, but annotated with a function execution space specifier. For example,

```
__global__ void my_kernel(.. ) { .. }
```

declares a kernel `my_kernel` that are callable from the host but executed on the device. In addition, the following specifiers are available: **`__device__`**, for functions that are called from and executed on the device, and **`__host__`**, for functions that are called from and executed on the host. We will only be concerned with functions called from the host and executed on the device in this thesis.

When launching a kernel the user must specify at least two things, namely, how many threads to use and the arguments to the function. Optionally, one should also provide the amount of a special type of memory needed for the function. We will return to the memory hierarchy in Section 2.1.4.

Each thread then sequentially executes the instructions contained in the kernel code, but threads are executed in parallel.

Kernels are invoked from the host program by using the special kernel-execution syntax indicated by triple chevrons:

```
my_kernel<<< .. >>>( .. )
```

where the arguments in the parentheses are normal function arguments as usual. The arguments in the chevrons specify how to group the threads, which will be described in the following section.

Often the host program needs to wait for the device to finish in order to get the results of the computation. Forcing the host to wait can be achieved by the keyword **`cudaThreadSynchronize`**`()`. For example, if the result of one kernel, say `kernel_a`, is required by another subsequent kernel, `kernel_b`, we insert a barrier:

```
kernel_a<<< .. >>>( .. );
cudaThreadSynchronize();
kernel_b<<< .. >>>( .. );
```

which effectively lets the host wait for all threads in `kernel_a` to terminate before the host program continues.

2.1.3 Thread Hierarchy

When specifying the number of threads the user must also specify how to group these threads. We will call this grouping or structuring for the thread space.

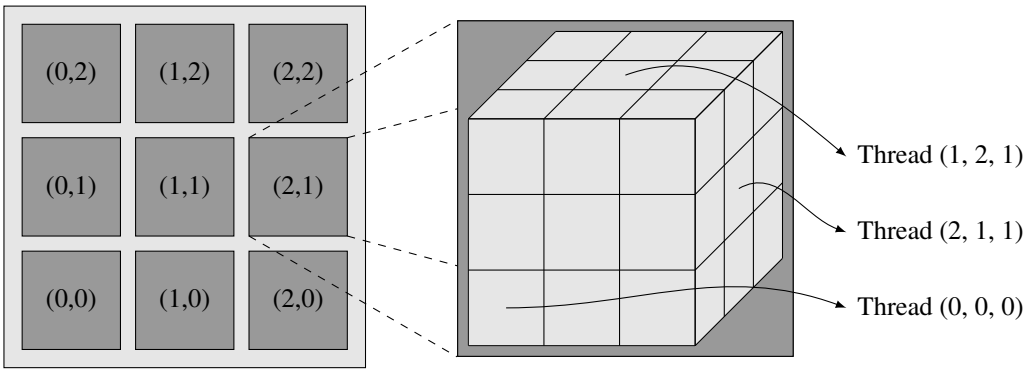


Figure 5: Two-dimensional grid (left) with a three-dimensional thread block (2,1) (right), along with individual threads in that block.

The thread space is split into three categories, namely threads, blocks, and grids, where each category is enclosed in its successor. This is shown in Figure 5.

At the lowest level, we have threads which are organized in thread blocks or just blocks. Thread blocks have, as their name suggests, three dimensions, x, y, and z, and all dimensions must be greater than or equal to one. Next, a collection of blocks are organized in a grid. Completely analog, grids are three-dimensional (although slightly confusing, as grids usually refer to sets of evenly spaced parallel lines at particular angles to each other in two dimensions). Finally, we have a kernel which, as mentioned above, must specify the thread space, i.e., grid and block size. Thus, we have the mnemonic: *a kernel is executed in a grid of blocks of threads*.

The kernel-execution syntax takes two arguments between the chevrons, the first specifying the grid dimensions and the second specifying the block dimensions.

Each thread launched has access to two three-dimensional vectors corresponding to its conceptual place in the block and the blocks place in the grid, respectively. The thread and block vectors can be accessed through the built-in functions `threadIdx` and `blockIdx`, respectively, and the vector elements can be accessed by specifying the dimension. For example, `threadIdx.x` gives the threads position in the blocks x dimension, and `blockIdx.x` gives the blocks position in the grids x dimension. Similarly, each thread can access the sizes of the dimensions for both blocks and grids, e.g., `blockDim.x` and `gridDim.x` for the length of the blocks and grids x dimension, respectively.

In current architectures, the size of thread blocks are restricted to 1024 threads no matter its dimensions, but multiple equal-sized blocks can be launched.

Furthermore, blocks are required to execute independently, hence the code is scalable with hardware. But analogously to how the host may be forced to

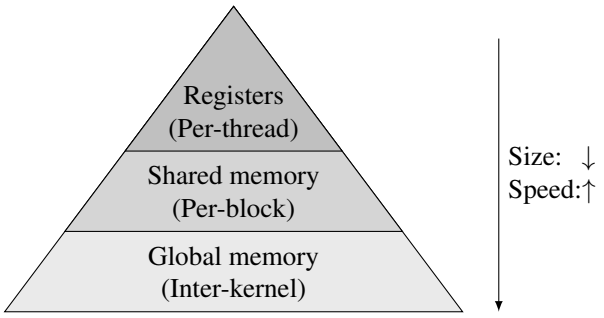


Figure 6: Memory hierarchy in the CUDA programming model. Actually, there is also thread-local global memory that are limited in size and are as slow as global memory.

wait for all threads in a kernel to terminate, we may also force all threads in a thread block to wait for all threads in the block to reach a barrier. Such a barrier is issued with **__syncthreads**(), and can be thought of the block-level equivalent of **cudaThreadSynchronize**(). When using **__syncthreads**() one must ensure that all threads are actually reaching this barrier at some point, otherwise, the kernel will deadlock.

2.1.4 Memory Hierarchy

The memory hierarchy in CUDA can be illustrated using a traditional pyramid hierarchy, with the slowest and largest type of memory at the bottom and the fastest and scarcest type at the top. Roughly, we have three main categories ranging from slowest to fastest, namely global memory, shared memory, and registers. This is shown in Figure 6.

The large and slow global memory must be allocated and initialized by specialized CUDA functions, e.g., **cudaMalloc**() and **cudaMemcpy**(), which allocates memory on the device and copies memory between the host and the device, respectively. The duration of global memory is inter-kernel meaning that it is persistent across kernels launched by the same application, and thus it can be used to communicate between threads in different grids. Global memory is free'd by calling **cudaFree**().

Shared memory, which can be thought of as a software-managed cache is much faster than global memory and is allocated on a per-block basis, i.e., threads from the same block access this same shared memory when allocated. The lifetime of shared memory is the same as the block, thus it can only be used to communicate between threads from the same block. Shared memory is allocated either statically by specifying the amount of memory in the kernel code or dynamically using the kernel-execution syntax.

The fastest type of memory is registers and they are private to each thread. Registers are very scarce and are used for storing variables, but one should be careful as the compiler may spill them to thread-local memory which is as slow as global memory.

Finally, the CUDA programming model assumes a weakly-ordered memory model. This means that the order in which data are written to global or shared memory, and others, may not be the same order in which other threads observe the data being written. Such reordering is generated by the compiler or by the CPU at runtime and may be used for optimizing bandwidth utilization. In a single-threaded program this does not matter – in fact, it happens all the time. For multithreaded programs, like GPU kernels, this can cause erroneous programs. The CUDA programming model provides memory fences to force an ordering on memory accesses. For example, `__threadfence()`, which guarantees that all memory accesses that appear in the code before the fence will also be observed, by other threads in the same thread block, as being executed before all accesses that occur after the fence. Note, that the observing threads must observe actual memory and not cached versions, thus the memory location should be declared volatile using the **`volatile`** keyword.

The reason is that the compiler will try to optimize memory accesses as long as it respects memory ordering semantics and visibility imposed by the memory model, synchronization functions, and memory fences. By declaring a memory location volatile the compiler is forced to give up all optimizations. In other words, **`volatile`** tells the compiler that the value may be used at any time by other threads and therefore any reference must be translated into an actual read or write from memory.

2.1.5 Atomic Functions

Because blocks may be distributed to different multiprocessors and execute concurrently, we cannot be sure of the ordering of the execution of threads. Thus, if we have a memory location that at least two threads are trying to access it may be the case that both threads read the same value, modify it, and write it back, which may result in an incorrect value, e.g., imagine you have a counter. This is a classic race condition.

CUDA provides a handful of atomic functions – atomic in the sense that they perform a read-modify-write sequence without any other thread being able to access the memory location. We will use three of these functions, namely:

```
atomicAdd(int* address, int val)
```

which reads the value at memory location `address` and stores it in `old`, adds

(a) CPU. **(b)** GPU.

Figure 7: CPUs are about control while GPUs are about throughput.

`old` and `val` and writes it back to `address`.

```
atomicCAS(int* address, int compare, int val)
```

which reads the value at `address` and stores it in `old`, performs the operation `old == compare ? val : old` and writes the result to `address`.

```
atomicExch(int* address, int val)
```

which simply reads the value at `address` and stores it in `old` before writing `val` to `address`.

Note, that these functions return the value that resided at `address` before the atomic operation. Only for **`atomicCAS`** and **`atomicExch`** will we use the returned value.

Note, that atomic functions do not impose synchronization nor do they work as memory fences. Furthermore, one might note that not all architectures have atomic functions, but we expect that when CUDA introduces atomic functions other vendors will do so too.

2.2 GPGPU Architecture

Because GPGPU programming is highly sensitive to the specific architecture at hand we will now look at how a GPU is actually implemented. We will use NVIDIAs GTX 780Ti as a vehicle but the ideas will be general.

CPUs perform well on complex programs containing a lot of branching structures because they are able to hide latency by speculative- and out-of-order execution. This idea is illustrated in Figure 7(a): There are a few ALUs to handle the actual computations and a beefy control logic to handle complex programs.

While CPUs are about control, GPUs are about throughput. They are optimized for problems that are data-parallel. That is, each thread can compute a part of the problem independently of the others while still be executing the same

operations. Furthermore, GPUs perform best if the ratio of arithmetic operations to memory operations are high. This is shown in Figure 7(b) where a lot of ALUs share some small control logic.

The GTX 780Ti consists of 15 *Streaming Multiprocessors* (SM). Each SM then consist of 192 *Streaming Processors* (SP), 64 double-precision units, 32 Special Function Units (for approximating specific functions), and 32 Load-Store Units. For this exposition it is sufficient to focus on SMs and how they execute threads at a high level.

When a CUDA enabled program executes on the host and launches a kernel, the blocks of the grid are distributed to SMs with available resources, i.e., threads belonging to the same block executes on the same multiprocessor. Multiple blocks may execute concurrently on one multiprocessor if resources are available, and as all threads from a block terminate new blocks are distributed to the SM if resources are available.

The number of blocks a multiprocessor can hold at the same time depends on how many resources, such as registers and shared memory, each block uses. Thus, you may have an incentive to use as few resources as possible in order to fit more blocks onto each multiprocessor. Recent NVIDIA GPUs may hold at most 16 blocks at the same time, though.

Today most GPUs use an execution model called *single instruction multiple thread* (SIMT), which can be seen as a development of the *single instruction multiple data* (SIMD) model. In the SIMT model the programmer writes a program that may contain branches and the like, hence threads can be seen as independent units. A SIMD type computer have multiple processing elements that each performs the exact same operation on multiple data elements simultaneously.

Although being conceptually independent units, threads on NVIDIA GPUs are executed in groups called *warps*. Warps consist of 32 threads that execute in lockstep, i.e., all threads execute the same instruction. Threads are deterministically divided into warps, with warp zero containing threads zero through, and including, thread 31, warp one containing threads 32 through, and including, thread 63, and so on.

2.3 Obtaining Good Performance on GPUs

In this section, we discuss ways to optimize GPU code, or how to avoid performance penalties if you will. As mentioned in the previous section, this is needed because kernels are hardware sensitive in a different way than programs for CPUs. Thus, this section can be seen as utilizing what we have learned in the previous two sections.

```
if(gid % 2 == 0) {
  ...
} else {
  ...
}
```

(a) Results in thread-divergence.

```
if(warp_id % 32 == 0) {
  ...
} else {
  ...
}
```

(b) Avoids thread-divergence.

Figure 8: Note that the two examples do not result in the same behavior, but are merely examples of thread divergence behaviour.

2.3.1 Thread divergence

We have previously talked about how threads are bundled in warps and executed in a lockstep-fashion, and how this relates to the SIMT model. This behavior has significant performance implications when the kernel code contains branches.

Consider the branch in Figure 8(a). Half the warp enters the true-branch and the other half enters the false-branch. Because all threads must execute the same instruction, all threads must actually first enter the true-branch. But threads for which the predicate evaluated to false is disabled, and the instructions are effectively *no operations* (NOOP)s. The same is the case for the false-branch.

The problem is that only half the threads will be active at any point through the if-branch, hence we are not fully utilizing instruction issuing. Therefore, when possible, we should write kernels that avoid intra-warp thread-divergence such that all threads in a warp follow the same execution path. An example is given in Figure 8(b).

2.3.2 Coalescing

More than often we encounter problems for which speed of computation rely on how fast we can read and write memory. Such problems are said to be memory-bound. The reads and writes to memory goes through a wide bus, usually transferring 512 consecutive bits, corresponding to 16 32-bit words, in one go. Thus, if threads in a warp access adjacent memory locations we need only two memory transfers to satisfy the access of all threads in a warp. This kind of access pattern is called coalesced memory access. In the worst case, all threads in a warp are accessing memory locations with no memory location within the same 16 words, such that we need 32 memory transfers to satisfy all threads.

2.3.3 Occupancy

In Section 2.2 we saw how GPUs are built for high throughput and that they are optimized for handling thousands of threads. So in order to utilize the hardware we should always have a lot of threads in flight, otherwise each SM will not have enough warps to hide latency between dependent instructions.

This idea is named *occupancy* and denotes the ratio of active threads on an SM to the maximum number of active threads supported by the SM. The maximum number of active threads is limited by resources available for the SM, e.g., registers and shared memory, and by a hard limit. For example, the GTX 780Ti supports 2048 active threads per SM as the hard limit.

Say you have a thread block of 256 threads that uses 16KB of shared memory, and that the total amount of shared memory is 48KB. Then you are able to fit three thread blocks on one SM, but the resulting occupancy is only $\frac{3\times256}{2048} = 37.5\%$. If instead you rewrite the code to use only 12KB then the occupancy is $\frac{4\times256}{2048} = 50\%$. In both cases, you are limited by shared memory resources.

On the other hand, imagine you have a thread block of 1024 thread using global memory instead of shared memory. Then you can fit two thread blocks on each SM and achieve an occupancy of 100%. But because thread blocks are using the much slower global memory the kernel does not necessarily run faster.

Occupancy can provide good guidance but should be used carefully.

2.4 Futhark

Futhark is a data-parallel programming language that introduces a programming model that builds on the vocabulary of functional programming by using second-order array combinators (SOACs). It is eagerly-evaluated, statically typed, purely functional, and has call-by-value semantics. The primary target of the optimizing compiler is efficient OpenCL code for execution on GPUs, but other targets are supported such as C or Python. This section builds on [Hen17, EH18, Hen, HSE+17].

2.4.1 Programming Model

The best way to explain the programming model of Futhark is to gain intuition for the language by example. Consider the following program that adds two to each element of an integer array (a historical note: this was the first Futhark program to be correctly compiled to OpenCL and executed on a GPU [EH18]):

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

This showcases the **map** construct which is probably the most important building block in Futhark. It takes as input a function $\alpha \to \beta$ and an array of values of type α, and produces an array of values of type β. This makes **map** very versatile as the user must only be concerned with the types while the compiler handles the parallelism.

Futhark employs a type of parallelism called explicit data-parallelism. It is explicit because the user through language constructs such as **map**, instructs the compiler of where to find the parallelism. It is data-parallel because the same operation is applied to different pieces of data. The example above is a prime example in understanding parallel functional languages.

Now, consider the following example that finds the largest absolute difference between elements in two integer arrays:

```
reduce max 0 (map abs (map2 (-) xs ys))
```

where **reduce** takes as input an operator of type $(\alpha \to \alpha \to \alpha)$, the neutral element of the operator of type α, and an array of values of type α, and produces a value of type α. One might think that this needs at least three traversals of the input arrays: first, subtracting the input arrays; second, finding the absolute value of the differences; third, finding the maximum of the absolute differences. In practice, all three operations will be fused such that arrays `xs` and `ys` will be traversed only once in global memory [HLO16].

This is the primary strength of Futhark: The compiler takes the responsibility of leveraging the compositionality of inherently-parallel bulk operators, such that the user does not have to, although some experience with Futhark may help you write faster programs.

2.4.2 Language features

Below we present language features of Futhark that are important in order to understand some of the design choices we made and that we will see later in this thesis:

Regular arrays Futhark supports only regular arrays, i.e., nested arrays must have the same shape. Irregular arrays can be expressed by a flat representation along with a flag array indicating the start of each inner array also called a segment. There already exists a library for segmented operations.

Size annotations Often you see yourself implementing some function on arrays that does not make sense for arrays of unequal length. In Futhark you can impose size constraints on the inputs using size parameters. Assume you

are writing a function for zipping two integer arrays and decide that it only makes sense for arrays of equal length:

```
let zip_int_arrays (xs : []i32) (ys : []i32) =
  map2 (\x y -> (x,y)) xs ys
```

Then you can require your function to accept only arrays of equal length by annotating the arrays with a size parameter `[n]`:

```
let zip_int_arrays [n] (xs : [n]i32) (ys : [n]i32) =
  map2 (\x y -> (x,y)) xs ys
```

Now the compiler will complain if the observed lengths of `xs` and `ys` are unequal. In fact, this is what happens if you type the first code snippet into the Futhark interpreter, `futharki`, because **`map2`** uses this strategy.

Parametric polymorphism Continuing the example of zipping integer arrays, and say you want to zip an array of floats. Then you would have to write a new function for floats. This duplication is never desirable, and Futhark solves it by using parametric polymorphism, which lets you use a type parameter, `'a`, to write polymorphic functions:

```
let zip_same_type 'a [n] (xs : [n]a) (ys : [n]a) =
  map2 (\x y -> (x,y)) xs ys
```

In fact, you could introduce two type parameters to zip two arrays of different types.

In-place updates Futhark supports in-place updates in order to avoid excessive copying of arrays. Staying in the histogram world, consider the following example from [EH18], and imagine you want to increment the count of a bucket:

```
let xs = xs with [i] <- xs[i] + 1
```

i.e., we update the array `xs` on index `i` with the value `xs[i] + 1`. By using an in-place update we avoid copying the whole array, and instead write only the to-be-updated element, an operation time proportional to the elements being updated.

Since Futhark is a functional language referential transparency must be preserved. Thus, in order to do in-place updates, the compiler must know

that no references to `xs`, or any variables aliasing it, exists after the in-place update. This is handled by a uniqueness type system described in the following.

Uniqueness types Continuing the example from before, consider the following function:

```
let increment (xs *[]i32) (i : i32) : *[]i32 =
  xs = xs with [i] <- xs[i] + 1
```

i.e., we are returning `xs` but with index `i` incremented by one. The asterisk in the parameter declaration for `(xs *[]i32)` tells the compiler that `xs` or any of its aliases must not be referenced after the call to `increment`. Note, that the return value has also an asterisk, which means that it shares no values with any visible variables.

2.4.3 Array Operators

When we introduced the programming model we showed how certain built-in functions are used as composable building blocks. To gain some intuition for these building blocks and in order to use some of them later, we here present some of the most important. In the following, we assume xs to be an array which elements are denoted $x_1, x_2, .., x_n$.

- **iota** : $(\texttt{n} : \texttt{i32}) \rightarrow \texttt{[n]i32}$
  iota $n \equiv [0, 1, ..., n-1]$
  Produces an array of integers from 0 to $n-1$.

- **replicate** : $(\texttt{n} : \texttt{i32}) \rightarrow \beta \rightarrow \texttt{[n]}\beta$
  replicate $n\ x \equiv [x_1, x_2, ..., x_n]$
  Produces an array with the value x repeated n times.

- **map** : $(\alpha \rightarrow \beta) \rightarrow \texttt{[]}\alpha \rightarrow \texttt{[]}\beta$
  map $f\ [x_1, x_2, \ldots, x_n] \equiv [f(x_1), f(x_2), ..., f(x_n)]$
  The function f is applied to every element in the array xs. This has inherently data-parallel semantics, as each element can be processed simultaneously.

- **reduce** : $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \texttt{[]}\alpha \rightarrow \alpha$
  reduce $f\ ne\ [x_1, x_2, \ldots, x_n] \equiv f(...f(f(ne, x_1), x_2), ..., x_n)$
  All values in the array is combined (reduced) using the binary operator f with ne as start value. We require that f is associative and that ne is the neutral element for f.

- **scan** : $(\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to []\alpha$
  scan $f\ ne\ [x_1, x_2, \ldots, x_n] \equiv$
  $[f(ne, x_1), f(f(ne, x_1), x_2), ..., f(...f(f(ne, x_1), x_2), ..., x_n)]$
  Produces an array of the same size as xs, by combining all prefixes. This corresponds to applying **reduce** to all possible prefixes of xs, and return an array of values. Like for **reduce** we require that f is associative and that ne is the neutral element for f.

- **stream_red** : $(\beta \to \beta \to \beta) \to ([]\alpha \to \beta) \to []\alpha \to \beta$
  stream_red $g\ f\ [x_1, x_2, \ldots, x_n] \equiv g(...g(g(f(x_1), f(x_2)), ..., x_n)$
  Assume a function `distr_p` that, when given an input array, chunks this array into an array of p subarrays of equal length (except possibly the last subarray). **stream_red** will apply `distr_p` to xs to obtain p chunks. Then f will be applied to each of the chunks in parallel, and ultimately reduce the subresults to one result by applying g. We require that the combining function g is associative.

map, **reduce**, **scan** belongs to a category of functions which in the functional programming world is called second-order array combinators, or SOACS. As previosly mentioned the goal is to provide a few of these functions with efficient parallel implementations and effecient optimisations, such that they can be used as building blocks.

The function **stream_red** is more exotic and derserves some extra explanation. The high-level idea of **stream_red** is based on following semantic equivalence (written here with **reduce**):

$$\textbf{reduce}\ \oplus\ e_\oplus\ xs\ \equiv \textbf{reduce}\ \oplus\ e_\oplus\ (\textbf{map}\ (\textbf{reduce} \oplus e_\oplus)\ (\texttt{distr}_\texttt{p}\ xs))$$

which says, that we can rewrite a reduction as mapping the reduction function over chunks of the input array, before reducing the intermediate results. In this way the compiler can use the chunking function, $\texttt{distr}_\texttt{p}$ to control the amount of parallelism needed depending on the situation. We will see later how this can be used to compute histograms.

3 Problem Statement and Related Work

Futhark relies on having a few efficient building blocks that can be combined to form solutions to complex problems. Now and then we encounter problems that cannot be efficiently solved using this approach. One way go about this is to create a new SOAC, although this should be considered carefully as it requires work through the whole compiler pipeline as well as maintenance.

In this chapter, we first explore on a high level the strengths and weaknesses of different approaches to computing general reductions. To justify our implementation effort we demonstrate how these approaches can currently be implemented in Futhark. Next, we show how the existing **`scatter`** construct can serve as an inspiration for the new language construct. Finally, we give a brief outline of the existing research, including solutions specific to histogram computations, as well as libraries and languages.

3.1 High-level Strategies

This section investigates the work complexities of different strategies for computing histograms in a parallel setting. We will use W to denote the work complexity of an algorithm, i.e., the total number of operations performed by the algorithm and we will use D to denote its depth complexity, i.e., the longest sequential dependency chain. Furthermore, we say that a parallel algorithm is work efficient if it does not perform asymptotically more operations then the sequential algorithm, in this case $O(N)$.

In the following let `N` be the number of input elements, `H` be the number of buckets in the histogram, `f` be a bucket function computing an index, `ind`, and a value, `val`, when applied to an element from the input array, `img`. The index and the value is then used for updating a histogram, `histo`.

We will use the notation `forall` to mean a parallel loop which indices are distributed among hardware threads, and `forseq` to mean a sequential loop where each index is computed in sequence by the same hardware thread. Fur-

```
forall(ii = 0; ii < N; ii++) {
  forseq(i = 0; i < H; i++) {
    histo[i] = 0
  }
  (ind, val) = f(img[ii])
  histo[ind] += val
}
// Reduce N histograms (determined by parallel loop).
```

Figure 9: Data-parallel solution. Each thread initializes its own subhistogram and processes one input element. This results in N subhistograms to be combined.

thermore, let the number of software threads launched be $T := \min(\text{HDW}, N)$, where HDW is the number of hardware threads needed to saturate the machine. Finally, we assume that subhistograms are reduced using a work efficient segmented reduction.

The simplest approach in order to avoid collisions is to let each thread compute its own histogram; this is a data-parallel solution. In addition, and for demonstration purposes, we let each thread process only one input element. Pseudo-code for this strategy is shown in Figure 9, where the outermost loop is distributed among N threads, and each thread then computes the innermost loop of size H and the following to statements. Its work complexity is then:

$$W_{data-parallel} := O(N \times H + N)$$

In comparison, the sequential algorithm performs $O(N)$ operations, thus this solution is not work efficient.

One way to reduce the amount of work is by distributing the outer loop among fewer threads, effectively letting each thread process multiple elements. Thus, as an improved data-parallel strategy, we let each thread process a chunk of elements of size N/T, reducing the number of threads needed to $N/(N/T) = T$. This is shown in Figure 10, and its work complexity is:

$$W_{data-parallel-chunk} := O\left(T \times H + T \times \tfrac{N}{T}\right) = O(T \times H + N)$$

such that when $T \times H$ is comparable to or a fixed multiple of N, we have:

$$W_{data-parallel-chunk} = O(k \times N + N) = O(N)$$

and the algorithm is work efficient. In particular, if we have $N/T = H$ then we get $O(N + N) = O(N)$.

```
forall(ii = 0; ii < N; ii += (N/T)) {
  forseq(i = 0; i < H; i++) {
    histo[i] = 0
  }
  forseq(i = ii; i < min(ii + (N/T), N); i++) {
    (ind, val) = f(img[ii])
    histo[ind] += val
  }
}
// Reduce N / H histograms (determined by parallel loop).
```

Figure 10: Improved data-parallel solution. Each thread initializes its own subhistogram and processes a chunk of input elements of size H. This results in N/H subhistograms to be combined.

```
forall(i = 0; i < H; i++) {
  histo[i] = 0
}
forall(i = 0; i < N; i++) {
  (ind, val) = f(img[i])
  atomic_operation{ histo[ind] += val }
}
// Only one histogram -> no reduction needed.
```

Figure 11: Non-data-parallel solution. Each thread processes one element but all threads cooperate on the same histogram.

Furthermore, consider the situation where we have atomic operations, such that threads may cooperate on one histogram. This is at the other extreme as opposed to the first data-parallel solution where all threads had their own histogram. Figure 11 shows the situation where each thread processes one element and all threads cooperate on a single histogram. Obviously, this is work efficient, both with and without chunking. We may note, that in the case where all input image pixels maps to the same bucket, then its depth complexity is $O(N)$ because atomics serializes the whole computation.

Finally, we consider the sort-reduce strategy shown in Figure 12. The map is work efficient for obvious reasons. The `sort_by_key` function can be implemented using radix-sort which is known to have work complexity $O(k \times N)$. Thus, when k is 32-bits, its work complexity is $O(N)$, and so, in this case, it

```
let (xs,  keys)    = map f img
let (xs', keys')   = sort_by_key(xs, keys)
let (xs'', keys'') = reduce_by_key(xs', keys')
```

Figure 12: Functional-style pseudo-code for computing a histogram.

is work efficient. The `reduce_by_key` function can be implemented as a segmented reduction, which is known to be asymptotically work efficient. Thus, the combination of the three is work efficient, but it is not really efficient in practice because of the potential high overhead of sorting. Furthermore note, that in the case where not all buckets are present in the input this does not produce the same result as the other solutions above.

This constitutes the landscape. In Chapter 4 we will pursue a combination of the data-parallel-chunk approach and the atomic approach. But first, we will see how to implement the data-parallel-chunk and the sort-reduce strategies.

3.2 Current State for Histograms in Futhark

The justification for this thesis is that we claim that currently there is no scalable way of efficiently expressing generalized reductions. In this section, we will look at some of these solutions, which are from [Hen]. For the sake of simplicity, we will look at the specific case of histogram computations, and we assume that indices are not out of bounds.

One might be tempted to write the sequential solution directly as follows:

```
let histo_seq (n : i32) (is : []i32) : [n]i32 =
  loop acc = (replicate n 0) for i in is do
    let acc[i] = acc[i] + 1
  in acc
```

While this produces the correct result it is simply too slow in practice as it is not possible to extract the parallelism, i.e., it will be sequential.

Another solution is to utilize the previously mentioned efficient building blocks. Futhark provides what is called streaming combinators, which assigns a function to some number of threads and then combines the per-thread results into a final, single result:

```
let histo_stream (n : i32) (is : []i32) : [n]i32 =
  stream_red_per (map2 (+)) (histogram_seq n) is
```

Here we apply `histo_seq` as the per-thread operator and combine results using **`map2`** `(+)`. This might be fine for a small number of buckets, but a problem for a large number of buckets. This is so because each thread allocates its own private histogram, thus at some point we will not be able to launch enough threads to saturate the machine due to insufficient memory.

The final solution we will look at is not straight forward and is not very efficent in practice:

```
let histo_sgm_red (n: i32) (is: []i32) : [n]i32 =
  let num_bits = t32 (f32.ceil (log2 (r32 n)))
  let is'  = radix_sort num_bits i32.get_bit is
  let flags = map2 (!=) is' (rotate 1 is')
  in segmented_reduce (+) 0 flags is'
```

Here the indices are first sorted using a radix sort, before a segmented reduction is applied to sum the buckets on a per-bucket basis. This solution is very flexible but both the radix sort and the segmented reduction are quite expensive operations.

In total, we think that this justifies a new construct in terms of the implementation effort.

3.3 Extending scatter

In data-parallel computing, a scatter operation receives as arguments an original array, an array of indices, and an array of values and it updates the elements of the original array at the corresponding indices with the new provided values. One can generalize the scatter construct to represent a fusion between a map and a scatter, in which the mapped function transforms some arbitrary type α into index-value pairs of type $(\texttt{i32}, \beta)$, and the scatter updates the original array based on these index-value pairs. The advantage is that the fused map-scatter does not need to materialize the index-value arrays in memory.

Currently, the intermediate representation of **`scatter`** in the Futhark compiler has the following type signature:

$$\textbf{scatter} \ : \ [\texttt{n}]\beta \to (\alpha \to (\texttt{i32}, \ \beta)) \to [\texttt{n}]\alpha \to [\texttt{n}]\beta$$

An example of its semantics, assuming a tuple-of-array representation and type $\alpha = (\texttt{i32}, \ \beta)$, is given by:

$$\textbf{scatter}\ [\texttt{b}_0, \ldots, \texttt{b}_{\texttt{m}-1}]\ \texttt{id}\ [0, 2, 4, 6]\ [\texttt{a}_0, \texttt{a}_2, \texttt{a}_4, \texttt{a}_6]$$

which results in

$$[\texttt{a}_0, \texttt{b}_1, \texttt{a}_2, \texttt{b}_3, \texttt{a}_4, \texttt{b}_5, \texttt{a}_6, \texttt{b}_7, \texttt{b}_8, \ldots, \texttt{b}_{\texttt{m}-1}]$$

However, the current **scatter** has two problems. First, it has a nondeterministic semantics when there are several to-be-updated values corresponding to the same index, i.e., when the index array contains duplicates. For example, if the map results in $[(0, 3.0), (0, 4.0), \ldots]$ then the first element of the result array may be either 3.0 or 4.0 depending on which update gets executed first. Second, it lacks the ability to combine several values corresponding to the same index as it will simply overwrite that current element.

One way of thinking of a histogram construct would be to generalize the scatter construct with respect to the second problem by adding support for combining values on duplicate indices. The starting point is to generalize **scatter** as below:

$$\textbf{scatter'} \ : \ [\mathtt{n}]\beta \rightarrow (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow (\alpha \rightarrow (\mathtt{i32},\ \beta)) \rightarrow [\mathtt{n}]\alpha \rightarrow [\mathtt{n}]\beta$$

such that it, in addition, takes as input an associative and commutative operator, $(\beta \rightarrow \beta \rightarrow \beta)$, and identity element, β. The new **scatter'** would have the following data-parallel semantics:

1. The map produces, semantically, an array of index-value pairs.
2. This array is sorted with respect to indices.
3. The sorted array is then reorganized as a two-dimensional irregular array in which the segments correspond to values that share the same index.
4. An irregular segmented reduction, with a specified binary associative operator computes the to-be-updated value for each index.
5. Ultimately, each such index of the original array is updated to the previously computed combined-value.

The observant reader may notice that the new **scatter'** still suffers from the first problem as it will combine all values with the same index, i.e., it does not have an update semantics. For our purpose this is not a problem as **scatter'** will be deterministic with respect to the histogram semantics.

3.4 Brief Outline of Research

The release of CUDA in 2007 marks the start of huge amounts of work dedicated to efficient histogram computations on GPUs. We investigate selected papers presenting carefully crafted implementations specifically for histogram computations. Even though these solutions cannot easily be modified to perform

generalized reductions we can definitely learn from the ideas. In addition, we discuss two master's theses that, among other things, investigate possibilities for efficient computations of generalized reductions on GPUs. Finally, we look into two well-established existing solutions.

3.4.1 Histogram Specific Solutions

The papers we have looked at all identify two main performance bottlenecks when computing histograms, namely, random writes and collisions. The two problems are connected as we will see, but we will first look at the former.

Podlozhnyuk, who seems to be the first to release a paper on histogram computation in CUDA [Pod07], uses subhistograms in shared memory to mitigate the latency induced by slower global memory. Subhistograms are then written to global memory where they are either combined into a final histogram, often by a segmented reduction, or combined directly by using atomic operations. Shams & Kennedy [SK07], Nugteren et al. [NvdBCM11], Brown & Snoeyink [BS12] all follow his approach using subhistograms in shared memory.

Because shared memory is limited and because early solutions did not have shared memory atomics and therefore often used per-thread subhistograms, the main problem then becomes minimizing per-thread shared memory usage in order to increase occupancy. Next, the problem becomes avoiding shared memory bank conflicts and the studied solutions go to great extend to mitigate this. We will not go into details but merely state that this is one approach.

Instead, we present one proposal to show the nature of these types of optimizations. Shams & Kennedy compact histograms using four 8-bit counters per 32-bit word, such that each thread needs 64 words to store a 256 bin histogram. The per-block histogram is then partitioned such that each thread holds 4 32-bit buckets in registers. Flushing to per-block buckets are done for every 63 words per thread to prevent overflow. Since they use 64 threads per block that need 16KB of shared memory each, and that they are running on the Fermi architecture that has 48KB of shared memory, this allows 3 thread blocks per SM. As they note, this amounts to an occupancy of 12.5% which is half of [Pod07] but the double of [NvdBCM11].

We will not pursue the shared memory solution as extreme as this, squeezing out almost every bit, but, as we will see later, instead use atomic operations.

Solutions to the second performance bottleneck, collisions, depends on the fact that atomic functions were not available for shared memory until around 2011. Podlozhnyuk proposes a software implementation of atomic operations that later papers also use. Since collisions become expensive the problem is now to avoid them. This is either done by using per-thread subhistograms or per-warp subhistograms, which again amounts to avoiding bank conflicts. Most

papers recognize that collisions are more frequent in real image data. Nugteren et al. come with two alternative and mutually exclusive proposals. Their first proposal is to shuffle the data in global memory before processing as usual. The second proposal is to just not read in coalesced fashion in the first place.

We will rely heavily on efficient atomic operations and instead, reduce the number of collisions by using multiple subhistograms per-block. But they recognize that real image data may increase the frequency of collisions, which complicates an efficient solution.

Finally, Shams & Kennedy proposes a solution supporting an arbitrary number of buckets. They use a multi-pass algorithm which requires multiple passes of the input, as only a range of buckets is processed in each pass. Although this solution is very flexible, it seems to be inefficient as it depends on both the input size and number of buckets.

We will investigate the possibility of supporting as many 32-bit buckets as global memory allow.

3.4.2 Generalised Reductions and Similar Operations

In his master's thesis, Eilers [Eil14] makes and extensive investigation of the possibilities for efficient implementations multireduces on GPUs, where multireduce is simply another word for generalized reduction. This reflects the lack of common nomenclature in the area. His investigation includes both commutative and non-commutative operators, in both global and shared memory, arbitrary operators and an arbitrary number of buckets.

In addition to replicating most of the solutions discussed above he proposes a solution with multiple threads cooperating on subhistograms directly in global memory. It has not been possible to see the actual implementation, but he reports promising results, especially when compared to its simplicity.

This sounds like a viable solution for our purpose. Our solution must support arbitrary operators, and therefore also arbitrary data types. Thus a solution not using shared memory seems like a good starting point because we need not be concerned with memory usage. Furthermore, our solution must support a very large number of buckets. Finally, Eilers makes a small experiment on the number of subhistograms in global memory. We will try to develop a heuristic indirectly choosing the number of subhistograms by choosing the number of threads cooperating.

In a later master's thesis, Joekladal [Jø16] follows up on and somewhat extends Eilers's work on multireduces. Joekladal proposes a crude heuristic for the cooperation level in order to minimize shared memory usage. Because he is keeping subhistograms in shared memory the upper limit is 8192 32-bit buckets. This he solves by developing a discriminator operator, that partitions input ele-

ments, effectively grouping elements by their label before applying a segmented reduction on them.

We will not pursue such a sorting approach but instead, focus on cooperation on multiple histograms in global memory, and use a heuristic to choose the amount of cooperation in order to minimize contention.

3.4.3 Libraries and Languages

In addition to investigating techniques specifically for computing histograms we have also looked at how general purpose programming languages handle generalized reductions. In particular, we have looked at Thrust [thr], a parallel algorithms library resembling the C++ Standard Template Library, and at Accelerate [acc], a domain specific language embedded in Haskell for data-parallel array programming.

Accelerate

In Accelerate we have the `permute` function, which performs a scatter-like operation using a combining operator as shown in Section 3.3. It has the following type signature:

$$\begin{aligned}\texttt{permute} : &(\texttt{Exp a} \to \texttt{Exp a} \to \texttt{Exp a}) \\ &\to \texttt{Acc}\,(\texttt{Array sh' a}) \\ &\to (\texttt{Exp sh} \to \texttt{Exp sh'}) \\ &\to \texttt{Acc}\,(\texttt{Array sh a}) \\ &\to \texttt{Acc}\,(\texttt{Array sh' a})\end{aligned}$$

where the arguments are, in order: a combining function, an array of default values, a bucket function, and an array of input values. It produces an array that is initialised with the provided default values an where values are written using the combining function.

The documentation for `permute` provides a small example of computing a histogram:

```
1 let histogram :: Acc (Vector Int) -> Acc (Vector Int)
2     histogram xs =
3       let zeros = fill (constant (Z:.10)) 0
4           ones  = fill (shape xs)         1
5       in
6       permute (+) zeros (\ix -> index1 (xs!ix)) ones
```

Lines 3 and 4 initialise two arrays with zeros and ones, respectively. The arrays

are provided as arguments to the `permute` function in line 6, along with the addition-operation as the combining operator, and a bucket function.

We know from [McD15] that atomic operations in Accelerate are implemented using either a critical section protected by a spin-lock, needed due to Accelerate using an internal struct-of-arrays representation, or a compare-and-swap style. [1] Even though we have not been able to locate it in the source code, we expect that they recognize operators that have a corresponding atomic function, for example addition, and generate code that uses that function instead.

Furthermore, although we have not been able to determine this from the source code either, it appears that they do not use anything similar to subhistogramming in order to reduce the frequency of collisions.

In total, the semantics of `permute`, along with its style and flexibility as exposed to the user, is what we aim at.

Thrust

Thrust provides a function called **`reduce_by_key`**. Given an array of indices and an array of values, it performs what is semantically a segmented reduction. That is, it combines adjacent values using a combining operator only if the corresponding keys are equal, e.g., given the following arrays of keys and values:

$$\begin{aligned} \text{keys} &: [1,1,1,2,2,2,3,1,1] \\ \text{values} &: [1,1,1,1,1,1,1,1,1] \end{aligned}$$

it produces

$$\begin{aligned} \text{result keys} &: [1,2,3,1] \\ \text{result values} &: [3,3,1,2] \end{aligned}$$

Even though this behaviour is nicely described in the documentation for Thrust, it is not exactly what we epxect from the name of the function, as this is more of a segmented reduction. In particular, we would expect the result to be:

$$\begin{aligned} \text{result keys} &: [1,2,3] \\ \text{result values} &: [5,3,1] \end{aligned}$$

such that the last 1's in the keys array contributes the to the first bucket in the values array, which is the bucket for the other occurences of 1 in the keys array.

Nevertheless, in order to compute a histogram we can simply sort the values by indices first. Fortunately, the function `sort_by_key` does exactly that. For

[1] See `https://en.wikipedia.org/wiki/Compare-and-swap`.

```
 1 // sort keys
 2 thrust::sort(thrust::device,
 3              d_indices.begin(),
 4              d_indices.end());
 5
 6 // compute histogram
 7 thrust::reduce_by_key(thrust::device,
 8                        d_indices.begin(),
 9                        d_indices.end(),
10                        d_values.begin(),
11                        thrust::make_discard_iterator(),
12                        d_values.begin());
```

Figure 13: Histogram computation in Thrust.

a histogram it is unnecessary to sort the values as they are all ones, and thus we can use the faster **sort** function for sorting only the indices. Figure 13 displays the histogram computation in Thrust, and we assume having two arrays; an array of indices, `d_indices`; and an array of values, `d_values`, which are initialised with zeros. In lines 2-4 we take advantage of the fact that all values are one and so we need only to sort the indices. This is followed by a function call to **reduce_by_key** which, as described above, acts as a segmented reduction.

The behaviour of **reduce_by_key** leads us to believe that it is implemented as a kind of prefix-sum, although we have not been able to determine this from the source code.

On the other hand, sorting has its benefits, as it makes the implementation data independent, i.e., its runtime performance is stable across all input distributions. Despite this desirable feature, the above combination of a sort and semantically a segmented reduction seems expensive. Instead, we will pursue an implementation that does not need to pre-process data in order to reduce it.

Part II

Development, Implementation, and Benchmarks

4 Prototyping

The focus of this chapter is twofold. First, we investigate different strategies for atomic updates. This is due to a source array of tuples in Futhark is represented internally as two arrays, and for that reason, what appears as a single update of one array element actually corresponds to two updates in the core language; one for each array. In addition, since this project is aimed at a language implementation, we need to support any user-defined operator. Fortunately, CUDA and OpenCL provides a handful of atomic functions, some of which can be used to implement arbitrary atomic functions on any number of arrays.

The second aspect we consider is the unfortunate side effect of atomic functions, namely, serialization of simultaneous accesses to the same memory location. The impact on the performance of serialization can be reduced using the simple idea of subhistogramming. In order to determine the cooperation level we have run a comprehensive experiment, and based on the results of the experiment we propose a heuristic for choosing a cooperation level. The chapter is concluded by a summary of the results which we believe validates our heuristic.

The code presented in this chapter, although heavily modified for brevity and clarity, is available here:

`https://github.com/lolkat2k/masters-prototype/`

along with a setup for reproducing all presented results as well.

4.1 Strategies for Locking

In order to accommodate the needs outlined above with respect to user-defined operators, we investigate three strategies based on three different atomic functions, namely **`atomicAdd`**, **`atomicCAS`**, and **`atomicExch`**.

In this section, we assume, for simplicity, that we have already computed an index, `bucket`, which is never out-of-bounds, and that we perform the following operation:

```
histo[bucket] += 1
```

i.e., we are updating a bucket in a traditional histogram.

4.1.1 Addition Only

The simplest and fastest way to update the bucket is, basically, not a locking strategy because the addition is performed implicitly:

```
atomicAdd(&histo[bucket], 1);
```

It simply adds `1` to the memory location pointed to by `&histo[bucket]`. This is much faster than the other two strategies we present, thus if we can identify the user-defined operator as a simple addition we should choose this strategy.

4.1.2 Arbitrary Binary Operators on One Memory Location

The second strategy we present is based on the **atomicCAS** function. Before digging into the code we may note, although obvious, that **atomicCAS** does not implicitly perform the operation as opposed to **atomicAdd**. It merely compares the current value to a provided value, and depending on the truth value it either writes another provided value or the current value.

For this reason, we need a supporting structure for verifying that no other thread updated the current value in between our read, modification and write:

```
1 int old = histo[bucket];
2 int assumed;
3
4 do {
5   assumed = old;
6   old = atomicCAS(&histo[bucket], assumed, assumed + 1);
7 } while(assumed != old);
```

The code can be divided into three logical steps:

1. First, the current value is stored in line 5. The variable is named `assumed` because it contains the value we expect to be at the memory location when we perform our update. Thus, if the actual value has changed in the meantime, then our update is based on an incorrect value.

2. Second, we use **atomicCAS** to write a new value, which in this example is the result of the addition (line 6). If the current value is equal to what

we got in step one, the update is based on the correct value. If not, it has changed, and the updated value is based on an incorrect value.

3. Third, if the write in step two succeeded then stop. This is the case if the memory location contained what we had previosly read, thus the result of the addition is correct. Otherwise, go to step one (lines 4 and 7).

This strategy can be used for implementing all binary operators that work on a single memory location, i.e., internally the array must be represented by a single array. Because the supporting structure is implemented in software it is much slower than the hardware optimized addition presented above, but, as we will see, it is still much faster than the next strategy.

4.1.3 Arbitrary Operators on Multiple Memory Locations

In contrast with the previous, we here present a strategy that does not perform the computation implicitly, nor does it need to check whether it got the computation right. Instead, it relies on a critical section protected by a spinlock:

```
 1  int done = 0;
 2
 3  while(!done) {
 4    if(atomicExch((int *)&locks[bucket], 1) == 0) {
 5      // Critical section - start
 6      histo[bucket] = histo[bucket] + 1;
 7      // Critical section - end
 8      locks[bucket] = 0;
 9      done = 1;
10    }
11    __threadfence();
12  }
```

Here the array `locks` has length equal to the number of buckets, is initialised with zeroes, and is declared volatile. The atomic function **`atomicExch`** sets the provided value, 1, at the given memory location, `&locks[bucket]`.

This strategy has, besides the obvious, two interesting parts that need emphasis:

- In line 4 each thread attempts to acquire the lock for the given bucket. (The casting is necessary because the type of the memory location must be an integer pointer from the perspective of the atomic function.) If any thread attempts to acquire a lock that is already taken, it will set the value 1, but will also, receive a 1, and thus it will not enter the critical section. Only

threads receiving a 0 will enter the critical section, and only the thread currently holding the lock is able to release it.

- In line 11 we have the memory fence, which is basically ensuring that all threads trying to acquire the lock are also observing the correct order of each other's attempts. This is needed because atomic functions do not have memory fence nor synchronization properties.

Due to the critical section, we can update any number of arrays using any kind of operator. Thus, despite being much slower, it allows for greater flexibility than the other strategies, which seems like a fair tradeoff. In practice, we should only use this strategy as a fallback case when it is not possible to use one of the other strategies.

4.2 Strategies for Subhistogramming

Due to the expected negative impact on performance arising from the serializing effect of atomic operations, we investigate possibilities for mitigation based on the idea of subhistogramming. In addition, we recall that one of the performance bottlenecks identified in Section 3.4 was random writes. Thus, we implement this idea in both global memory and the much faster shared memory.

The implementation of this idea relies on precomputed values, but for clarity and brevity, we assume that all such values have been computed, and we will explicitly describe each value when they arise. Furthermore, both cases (global and shared memory) will be based on `atomicAdd` and chunking, i.e, one thread will process multiple input elements. We will still be updating a bucket in a traditional histogram. Finally, we assume that all excess threads have been guarded off.

4.2.1 Subhistogramming in Global Memory

The most straightforward way is to create subhistograms in global memory and let threads cooperate regardless of which threadblock they belong to:

```
1  // Global thread id.
2  int gtid = blockIdx.x * blockDim.x + threadIdx.x;
3
4  // Global histogram id.
5  int ghid = (gtid / coop_lvl) * his_sz;
6
7  for(int i=gtid; i<img_sz; i+=num_threads) {
8    int bucket = f(img[i]);
9    atomicAdd(&histo[ghid + bucket], 1);
10 }
```

Here we assume that the following values have been computed; the number of threads cooperating on one histogram, `coop_lvl`; the number of buckets in the histogram, `his_sz`; the number of input elements, `img_sz`; and the number of threads launched, `num_threads`.

The strategy has three main aspects to notice:

- The subhistograms are stored in global memory as one array, thus we need to compute an offset to find the correct subhistogram. Each thread computes the subhistogram to cooperate on based on its global thread index (line 5). This means that threads from different thread blocks may cooperate.

- We assumed that each thread computes a chunk of elements, which is expressed by the for-loop starting in line 7. Recall that coalesced reads are obtained if consecutive threads are reading consecutive elements. Therefore, we let each thread start reading the element corresponding to its global thread index, and the loop is incremented by the total number of threads. The loop terminates when the loop counter exceeds the number of elements in the input array.

- In line 8 we compute the bucket as if we had only one histogram, which allows us to reuse the bucket function independently of the number of subhistograms. The bucket is then used to obtain the correct index in the large subhistogram array in line 9.

Even though the above code has been simplified for brevity and clarity, it shows how simple it is to get a working implementation using subhistogramming in global memory. The hard part lies in tuning the cooperation level, which we will see later, but first, we show how to implement subhistogramming in shared memory.

4.2.2 Subhistogramming in Shared Memory

Compared to the global memory version it requires only a few more steps to implement subhistogramming in shared memory, but are not conceptually more difficult to understand. It can be divided into three logical steps: 1) initialize histograms in shared memory, 2) compute histograms in shared memory, and 3) copy histograms to global memory.

The implementation is displayed in Figure 14, and in addition to the variables from the previous section, we assume that the number of histograms that a thread block compute is given by `hists_per_block`. This value is determined by how many subhistograms that fit in shared memory, and the cooperation level. For example, if we can fit two subhistograms in shared memory, but the cooperation level is more than 512, then we can only compute a single subhistogram per block. On the contrary, if we can fit only one subhistogram in shared memory, then the cooperation level is equal to the thread block size (if we assume 32-bit buckets).

As said above, the code can be divided into three logical steps as also indicated in the code:

- In line 15 we create an array of integers in shared memory. The total size of shared memory is provided as an argument to the kernel and here we simply use all of it for a single array.

  The following loop in line 16 performs coalesced writes to initialize the subhistograms, which are completely analogously to the coalesced reading described in the previous section. Note, that we must use only threads from the current thread block because shared memory is only visible on a per-block basis.

  Immediately after the loop, we need a barrier, **`__syncthreads()`** to ensure that all buckets have been initialized before any thread starts working on any of the subhistograms.

- Histograms are computed completely analogously to the global memory strategy. The only difference is that we use a local histogram index in line 24 to compute the bucket. Following this block, we also have a barrier to ensure that no thread starts copying before all threads are done processing elements.

- Finally, the histograms in shared memory are copied to global memory. The important thing here is that each subhistogram in shared memory has a corresponding space in global memory that it is copied to. In line 12 we see that the global histogram offset is computed on a per-block basis.

```
1  // Local and global thread id.
2  int ltid = threadIdx.x;
3  int gtid = blockIdx.x * blockDim.x + ltid;
4
5  // Total number buckets for local histograms.
6  int his_sz_block = hists_per_block * his_sz;
7
8  // Local histogram id.
9  int lhid = (ltid / coop_lvl) * his_sz;
10
11 // Global histogram id.
12 int ghid = blockIdx.x * hists_per_block * his_sz;
13
14 // 1) Initialize histograms in shared memory.
15 extern __shared__ int shared_histo[];
16 for(int i=ltid; i<his_sz_block; i+=blockDim.x) {
17   shared_histo[i] = 0;
18 }
19 __syncthreads();
20
21 // 2) Compute histograms in shared memory.
22 for(int i=gtid; i<img_sz; i+=num_threads) {
23   int bucket = f(img[i]);
24   atomicAdd(&shared_histo[lhid + bucket], 1);
25 }
26 __syncthreads();
27
28 // 3) Copy histograms from shared to global memory.
29 for(int i=ltid; i<his_sz_block; i+=blockDim.x) {
30   global_histo[ghid + i] = shared_histo[i];
31 }
```

Figure 14: Subhistogramming in shared memory.

This offset is then used for copying into the correct subhistogram space in global memory. Note, that both the reads and the writes are coalesced.

We will see in a moment, that this strategy turns out to be faster than the one in global memory, indicating that the random writes have a significant impact on performance.

But this strategy is also more fragile. The obvious reason is that the amount of shared memory is limited, hence it cannot be used for all situations. One less obvious reason is that the more subhistograms we put in shared memory the fewer thread blocks can run on a streaming multiprocessor. In practice, this may or may not have a significant impact on performance, but we have not investigated the full implications of this strategy.

Note on `atomicExch`-based Strategy in Shared Memory

If we want to use the **`atomicExch`**-based locking strategy with subhistogramming in shared memory, we would straightforwardly translate the version using global memory:

```
while(!done) {
  if(atomicExch((int *)&locks[ghid + bucket], 1) == 0) {
    histo[ghid + bucket] +=  1;
    locks[ghid + bucket] = 0;
    done = 1;
  }
  __threadfence();
}
```

into one using shared memory, i.e., we would move `locks` and `histo` to shared memory, holding everything else equal. While the global memory version works, the shared memory version does not. We have not been able to figure out why the first version does not work in shared memory, as opposed to the second. Atomic functions do not impose constraints on memory orderings, and as such we are removing conditions (the threadfence) that should ensure other threads to observe the lock being released. We leave this issue for further studies and use the second version in the following.

Instead, we use the following strategy:

```
while(!done) {
  if(atomicExch((int *)&sh_locks[lhid + bucket], 1) == 0) {
    sh_histo[lhid + bucket] += 1;
    atomicExch((int *)&sh_locks[lhid + bucket], 0);
    done = 1;
  }
}
```

where `sh_locks` and `sh_histo` are shared memory versions of `locks` and `histo`, respectively.

4.3 Performance Experiment

In the previous, when we discussed subhistogramming strategies, we assumed that the cooperation level was given. In order to determine the cooperation level we have run a comprehensive study investigating the runtime performance of all three locking strategies for varying cooperation levels across a range of histogram sizes. Based on the results of this study, we present a heuristic for choosing a cooperation level, and this chapter is concluded by a summary of the experimental results which we believe justify our heuristic. The full experimental results are available in Appendix A

In addition, the experiment enables us to show further two things. First, we show the impact on performance of choosing the **`atomicCAS`** or **`atomicExch`**-based strategy compared to the **`atomicAdd`** strategy. Second, we show the impact on performance when using subhistogramming in shared memory compared to subhistogramming in global memory.

4.3.1 Setup

The experiment measures the runtime for computing a histogram from an input array of 10 million elements. For each histogram size we let the cooperation level range from no cooperation to full cooperation, i.e., from all threads computing their own histogram to all threads cooperating on the same histogram.

For each histogram size, the values of the input array elements are uniformly distributed and correspond to indices. Note, that this is the best case scenario as we minimize contention and thus serialization. The distribution of input data may have a significant impact on performance since both the frequency of collisions and the cache hit rate are affected. We leave this issue to further studies, but will revisit it briefly in the benchmarks.

All runtimes reported are averages of five runs and a single runtime measurement includes initialization and computation of subhistograms but not the final

reduction phase. The reduction phase for varying number and size of histograms are timed separately using the following Futhark program:

```
let main [H] (hists : [][H]i32) : [H]i32 =
  map (\col -> reduce (+) 0 col) (transpose hists)
```

It takes as input an unknown number of subhistograms all of size H, which can be seen as a matrix which rows corresponds to subhistograms. We then map a reduction over the rows of the transposed matrix, in which rows consist of the same bin from all subhistograms.

The final reduction phase is timed in Futhark because an efficient reduction kernel is non-trivial to implement by hand in CUDA and the fact that efficient code generation already exists in Futhark.

Finally, the performance of the code has been evaluated on a system consisting of an Intel Xeon E5-2650 CPU, and an NVIDIA GTX 780Ti GPU. The latter has 48MB of shared memory per block and 16MB of L1 cache. For NVIDIA GPUs with compute capability 3.x this split can be configured but 48MB of shared memory is the default and also the largest.

4.3.2 Heuristic for Choosing Cooperation Level

In order to minimize collisions but still keep the work complexity in check, we now propose a heuristic for choosing a cooperation level.

In Section 3.1 we discussed the work complexities of high-level strategies for computing histograms. Recall, that we had a data-parallel solution, Figure 9, where all threads compute their own histogram, and a solution where all threads cooperate on the same histogram using atomic functions, Figure 11. We observe that these two cases correspond to two extremes. Our heuristic then interpolates between these extremes by choosing a cooperation level. The optimal cooperation level was determined by analyzing the results of the experiment described above, and which are summarized in the following section.

As a theoretical justification, we here develop the work complexity for our heuristic. In the following, assume the number of software threads launched to be $T := \min(\text{HDW}, N)$, where HDW is the number of hardware threads needed to saturate the machine. Each thread computes a chunk of elements of size N/T. Now, let C denote the cooperation level we want to use, and let T/C be the number of subhistograms produced using that cooperation level. Then the work complexity can be computed as:

$$W_{heuristic} := O(\tfrac{T}{C} \times H + T \times \tfrac{N}{T})$$

Then, as per our assumption of uniformly distributed input data, we can allow up to H threads to cooperate on one subhistogram without collisions, i.e., $C := H$.

Substituting in this value we get:

$$W_{heuristic} := O(\tfrac{T}{H} \times H + T \times \tfrac{N}{T})$$

but in practice we that $HDW \ll N$, such that:

$$W_{heuristic} := O(HDW + N) = O(N)$$

meaning that subhistogramming in global memory is work efficient if we choose the cooperation level to be equal to the histogram size.

Thus, we have the following heuristic:

$$\begin{aligned} \text{Per-thread chunk} &:= \lceil \tfrac{N}{T} \rceil \\ \text{Cooperation level} &:= \ H \end{aligned}$$

which produces $\lceil T/C \rceil$ subhistograms. In practice, the runtime associatied with the final reduction, which depends on the number and size of subhistograms produced, is negligible when using our heuristic.

We note that the depth complexity is equal to N in the worst case, i.e., having one large histogram where all elements hit the same bucket. In practice, though, we expect to have depth $O(1)$, excluding the final reduction.

Finally, the above heuristic needs some adjustment when using subhistogramming in global memory: When only one histogram fit in shared memory and the cooperation level is greater than the block size, then the current heuristic would propose a cooperation level which is not possible. From the experimental results, we observed that setting an upper limit for cooperation level equal to the block size produces good results. The heuristic for subhistogramming in shared memory becomes:

$$\begin{aligned} \text{Per-thread chunk} &:= \lceil \tfrac{N}{T} \rceil \\ \text{Cooperation level (C)} &:= \min(H\ ,\ B) \end{aligned}$$

under the condition $\lceil \frac{B}{C} \rceil \times H \leq 12$KB. Here B is thread block size, and the condition ensures that at the size of at least one subhistogram, using 32-bit buckets and 48KB of shared memory, does not exceed the amount of shared memory. We then fit as many subhistograms as possible subject to two limitations, namely, the amount of shared memory and the the thread block size, and choose the smallest limit. When $H > B$ the work complexity will be greater than the one computed above, because fewer threads are cooperating on one histogram, causing the number of subhistograms to increase compared to the corresponding case in global memory. This again effects the the number of thread blocks needed to process all elements, but we expect this overhead to be negligible. In total, this leaves room for improvement of our heuristic regarding shared memory, but, as we will see, we obtain a significant improvement in runtime performance compared to global memory.

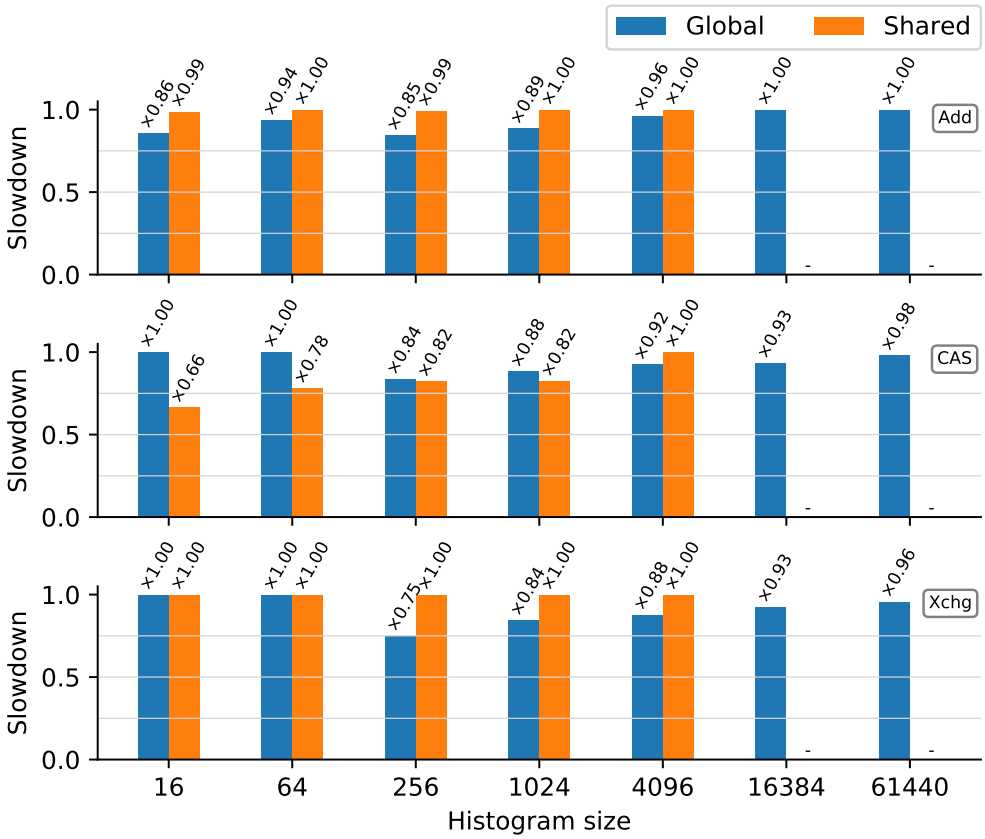


Figure 15: The impact on performance for subhistogramming when using our heuristic. For each histogram size, we find the fastest runtime, no matter its cooperation level, which is then used as a baseline when comparing the runtime obtained from using the cooperation level proposed by our heuristic. This is done for all locking strategies, indicated by **Add**, **CAS**, and **Xchg**, and memory spaces, **Global** and **Shared**. Values of 1.00 indicate that our heuristic proposed the cooperation level that produced the fastest runtime. See Table 3 for data corresponding to this graph.

4.3.3 Empirical Validation of Proposed Heuristic

The full experimental results are available in Appendix A and the proposed heuristic were chosen by analyzing them, but here, for clarity and brevity, we present a summary of the results which we believe justify the chosen heuristic.

Figure 15 shows the runtime performance of subhistogramming when using our heuristic. For each histogram size, the horizontal axis, we have compared the runtime performance obtained using our heuristic to the runtime for the co-

operation level giving the fastest runtime. The vertical axis then gives the slowdown, i.e., how did we compare to the fastest runtime, hence a bar with height 1.00 means that our heuristic provided the fastest runtime. This is done for both shared and global memory, and for each of the three locking strategies.

For **CAS** and **Xchg**, we see that for the two smallest histograms in global memory, 16 and 64, the cooperation level chosen by our heuristic provides the fastest runtime. For **Add**, on the other hand, is up to 14% slower, and the optimal cooperation levels in these cases are 16 times larger than the histogram size. This indicates that **Add** is implemented with specialized hardware that handles collisions better, and it suggests that it uses a higher cooperation level in this case.

For the two largest histograms in global memory, 16384 and 61440, we have the opposite pattern. Here our heuristic provides the optimal cooperation level for **Add**, but is within 7% for **CAS** and **Xchg**. For both **CAS** and **Xchg** the optimal cooperation levels are 16 times lower than the histogram size, i.e., 4096 and 16384, respectively.

For the medium-sized histograms in global memory, 256, 1024, 4096, and for all three operators, our heuristic does not provide the optimal cooperation level. The slowdown is up to 25%, while most slowdowns are around $8\% - 10\%$. From the raw data we see that this is due to the same patterns described above: **Add** allows for greater cooperation levels, while **CAS** and **Xchg** require lower cooperations. For the latter two operators, the data suggests that the cooperation level is equal to the histogram size divided by four.

For **Add** and **Xchg** on all histogram sizes in shared memory, our heuristic provides a close-to-optimal cooperation level. Only for **CAS** we see a significant decrease on small histograms in shared memory, and the raw data suggests we use a cooperation level equal to the histogram size divided by four. We have not investigated why **CAS** would perform significantly worse than **Xchg** in shared memory, and leave that for future studies.

In total, we believe that this justifies our heuristic for three reasons. First, for all three strategies, the heuristic is stable across a broad range of histogram sizes. Second, its simplicity is appealing in a language implementation, in which the small slowdowns seems tolerable compared to increased complexity in the compiler. Third, even though its implications are not yet fully understood it seems fairly easy to reason about, which is desirable in an already complex setting. Therefore, we will use the heuristic in our implementation.

Performance Penalty From Using Other Operators Than `atomiAdd`

Table 1 shows the impact on runtime performance from using **`atomicCAS`** or **`atomicExch`** compared to **`atomicAdd`**, when using our heuristic. For each his-

			16	64	256	1024	4096	16384	61440
Global	**Add**	(μs)	$434\mu s$	$484\mu s$	$744\mu s$	$913\mu s$	$947\mu s$	$913\mu s$	$997\mu s$
	CAS	$(\times)$	$0.39\times$	$0.40\times$	$0.33\times$	$0.24\times$	$0.20\times$	$0.20\times$	$0.21\times$
	Xchg	$(\times)$	$0.24\times$	$0.23\times$	$0.19\times$	$0.13\times$	$0.11\times$	$0.11\times$	$0.11\times$
Shared	**Add**	(μs)	$489\mu s$	$467\mu s$	$505\mu s$	$500\mu s$	$555\mu s$	-	-
	Cas	$(\times)$	$0.32\times$	$0.34\times$	$0.40\times$	$0.43\times$	$0.58\times$	-	-
	Xchg	$(\times)$	$0.19\times$	$0.20\times$	$0.21\times$	$0.21\times$	$0.19\times$	-	-

Table 1: The impact on performance of using **`atomicCAS`** or **`atomicExch`** compared to **`atomicAdd`**, for a range of histogram sizes (topmost row). For subhistogramming strategies in both global and shared memory, indicated by **Global** and **Shared**, respectively, row **`atomicAdd`** is used as baseline and displays runtime in microseconds; rows **`atomicCAS`** and **`atomicExch`** display slowdowns.

togram size, the topmost row, we use **`atomicAdd`** as baseline, row **Add**, which displays runtime in microseconds. We then compare the strategies **`atomicCAS`** and **`atomicExch`**, rows **Cas** and **Xchg**, respectively, to the baseline.

For small histograms in global memory, we see that **CAS** is up to $\frac{1}{0.39} = 2.56\times$ slower than **Add**, and that for larger histograms is up to $\frac{1}{0.20} = 5\times$ slower. This pattern repeats for **Xchg** which is $\frac{1}{0.24} = 4.16\times$ slower on small histograms and $\frac{1}{0.11} = 9.09\times$ slower on large histograms.

In shared memory, we see that for medium-sized histograms, which are the largest we have tried in shared memory, that the slowdown is less significant than in global memory.

In total, this suggests what we attempt to recognize the case when the operator is a simple addition and then utilize the corresponding atomic function. This also holds for **CAS**, i.e., if we can detect the case where we can use **CAS** instead of **Xchg**.

Speedup From Using Subhistogramming in Shared Memory

Finally, Table 2 shows the impact on runtime performance from using shared memory instead of global memory, when using our heuristic. For a range of histogram sizes we use the runtime using subhistogramming in global memory as baseline, and compare the runtime using shared memory.

For small histograms, it appears that we do not benefit from using the faster shared memory. One reason might be, that the impact on runtime performance from random writes is less significant for small histograms, i.e., the impact from

			16	**64**	**256**	**1024**	**4096**
Add	**Global**	(μs)	$434\mu s$	$484\mu s$	$744\mu s$	$913\mu s$	$1020\mu s$
	Shared	$(\times)$	$0.89\times$	$1.04\times$	$1.48\times$	$1.83\times$	$1.84\times$
Cas	**Global**	(μs)	$1103\mu s$	$1217\mu s$	$2235\mu s$	$3862\mu s$	$4296\mu s$
	Shared	$(\times)$	$0.72\times$	$0.89\times$	$1.79\times$	$3.33\times$	$4.51\times$
Xchg	**Global**	(μs)	$1812\mu s$	$2094\mu s$	$3948\mu s$	$6827\mu s$	$7495\mu s$
	Shared	$(\times)$	$0.70\times$	$0.88\times$	$1.67\times$	$2.93\times$	$2.50\times$

Table 2: The impact on performance of using shared memory compared to global memory, for a range of histogram sizes (topmost row). For each locking strategy **`atomicAdd`**, **`atomicCAS`**, and **`atomicExch`**, indicated by **Add**, **CAS**, and **Xchg**, respectively, row **Global** is used as baseline and display runtime in microseconds; row **Shared** display speedup.

an increased cache miss rate caused by distant writes is less severe in small histograms.

On the contrary, we see that with larger histograms we benefit from having subhistograms in shared memory. For example, for histogram size 4096 and **CAS** we see a 4.51 speedup, which will make the operator run in about $1000\mu s$, i.e., as fast as **Add** does in global memory. The runtime for **Xchg** can be more than halved, although it will still be slow compared to the other two. Finally, we can make **Add**, which is already very fast, run in almost half the time of global memory.

5 Implementation

This chapter describes the implementation of a new language construct in the programming language Futhark and its optimizing compiler. The code generation is based on the subhistogramming strategy in global memory described in the previous chapter.

The Futhark compiler supports both sequential and parallel code generation, and it therefore contains two pipelines: one for sequential programs and one for programs containing parallel parts. The focus of this chapter is the parallel pipeline generating host programs in C and calls OpenCL kernels, although other targets are supported. This pipeline is displayed in Figure 16 in a conventional three-stage structure, and this chapter describes our work progressing from front to back end.

In Futhark a pipeline comprises a collection of passes, each taking as input a program and producing a program as output. Our work has been focused on extending existing enabling transformations, i.e., passes that modify the abstract syntax tree to increase the effectiveness of subsequent passes, effectively modifying the whole pipeline to handle the new construct.

Finally, the Futhark compiler, including our contribution, is publicly available under a free software license at:

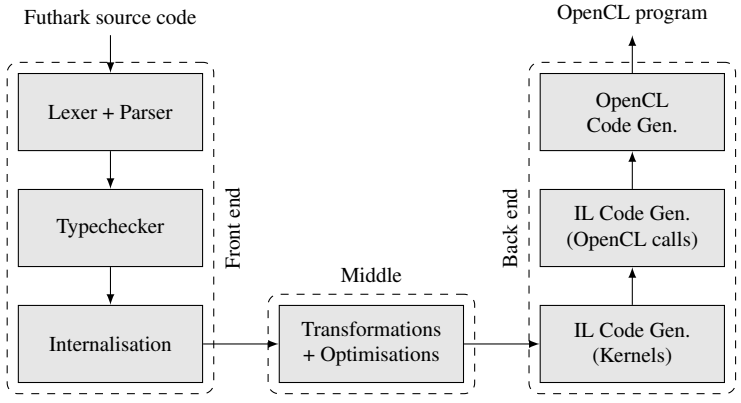
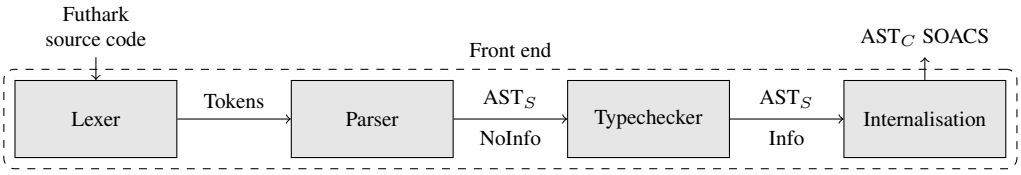


Figure 16: Data flow diagram for the GPU compilation pipeline.

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

The compiler is written in Haskell and we will reference files as Haskell modules, e.g., `src/Futhark/Representation/SOACS/SOAC.hs` is referenced as `Futhark.Representation.SOACS.SOAC`. To reference a specific function in a module the last component of a module reference will be the function name, which always start with a lowercase letter. Due to space issues and to increase readability we will sometimes omit code from listings, which will be indicated by `(..)`.

5.1 Front End



The abstract syntax for the Futhark language is split into two: a source version (AST$_S$) and a core version (AST$_C$). This increases flexibility in the middle phase of the pipeline as we will see later. In addition, the source version is parameterized by having type information or not, and the core version is parameterized by a representation.

With this in place, we say that the responsibility of the lexer and parser is to translate a Futhark source program into source abstract syntax with no type information. Then the type checker adds type information to the source abstract syntax, and, finally, the internaliser handles higher-order functions and modules, before translating the source abstract syntax into core abstract syntax parameterized by the SOACs representation.

5.1.1 Lexer, Parser, and Type Checker

We have dubbed the new language construct **`reduce_by_index`** and it will be exposed to the user through a library function from the SOACs library which is loaded by default through the prelude. Thus, users can simply use the **`reduce_by_index`** in their programs without loading any modules.

Figure 17 displays the function, where lines 1 through 3 constitutes the function declaration and line 4 is the function body. The function takes as input five arguments and it is important to notice the following: the first input, `dest`, and the result must be of equal length; the array of indices, `is`, and array of values, `as`, must be of equal length; and that the combining function, `f`, the element `ne`,

```
1  let reduce_by_index 'a [m] [n]
2    (dest : *[m]a) (f : a -> a -> a)
3    (ne : a) (is : [n]i32) (as : [n]a) : *[m]a =
4    intrinsics.gen_reduce (dest, f, ne, is, as)
```

Figure 17: The new language construct is exposed to the user through the SOACs library, which is also loaded by default through the prelude.

```
data ExpBase f vn =
  (..)
  | Apply (ExpBase f vn) (ExpBase f vn) (f Diet)
          (f PatternType) SrcLoc
  | GenReduce (ExpBase f vn) (ExpBase f vn) (ExpBase f vn)
              (ExpBase f vn) (ExpBase f vn) SrcLoc
```

Figure 18: `Language.Futhark.Syntax`.

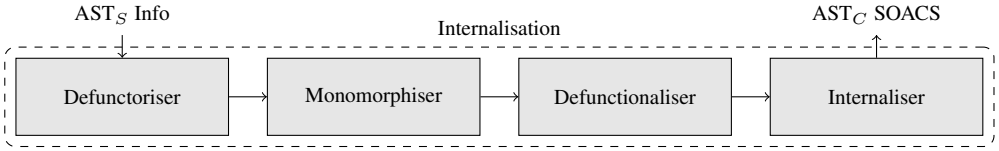
and the array of values must all be of the same type. It is the responsibility of the user to check whether `ne` is the neutral element for `f`.

Importantly, the body contains a single call to a function called `gen_reduce`, which takes all the same arguments as the library function.

We begin our presentation of the implementation right after the program has been lexed, parsed, and type checked. At this point, the program is represented by the source abstract syntax. The important parts of the syntax definition are shown in Figure 18, where the arguments to the data type are used to parameterize the abstract syntax by type information.

Because the construct is a higher-order polymorphic function, it will be wrapped by an `Apply`-node. The type checker has ensured that the constraints discussed above, imposed by the function declaration, are satisfied. Now, it is the job of the internaliser to translate the the `Apply`-node into a `GenReduce`-node.

5.1.2 Internalisation



Futhark supports both higher-order functions and modules, and in order to generate code for GPUs which has only limited support for function pointers, it

evaluates away modules and turn higher-order functions into calls to a first-order apply-function. The result of this process, a well-typed, monomorphic, module-free program in source abstract syntax is then transformed into core abstract syntax (as indicated by the small diagram below the section title) [Hov18].

The described process has been split into four separate passes; defunctoriser, monomorphiser, defunctionaliser, and internaliser. Our implementation has only modified the latter three, and therefore we will not go into further details about the defunctoriser than as to note, that it is responsible for removing modules. Thus, the defunctoriser receives a well-typed and polymorphic program with modules and produces an equivalent module-free program which is passed to the monomorphiser.

Monomorphiser

Since the the defunctionaliser works only on monomorphic programs, the job of the monomorphiser is to convert a polymorphic program into an equivalent but monomorphic program. That is, for each instance of a polymorphic function it specializes its type for that particular type instantiation determined by the application in the program. Effectively, it produces specialized functions from generic ones.

The above description is the task for the function in Figure 19. It takes as input a value of type `Exp`, which is a Haskell type synonym for `ExpBase` – the source abstract syntax – shown in Figure 18, and it produces a monomorphic expression inside the `MonoM`-monad.

As mentioned above, higher-order functions cannot be present in the abstract syntax at this point and are therefore wrapped in an `Apply`-node. `transformExp` then receives an `Apply`-node holding our construct, pattern matches on the number of arguments (line 5) and looks up if it knows a function named `gen_reduce` (line 6). Recall, that this function was the body of the library function in Figure 17. If these checks suceed, it produces a `GenReduce`-node (Figure 18) in source abstract syntax, and makes a recursive call to also transform its arguments (line 7 and lines 12-19).

Defunctionaliser

At this point, the monomorphiser has just turned all polymorphic functions into monomorphic functions, but the program still contains higher-order functions.

Higher-order functions are, as mentioned above, not well-suited for GPUs and one way to get around that is to turn them into first-order apply-functions [Hov18]. This is the job of the defunctionaliser, which main function is shown in Figure 20. The function takes as input a value in source abstract syntax, `Exp`, and

```
 1 transformExp :: Exp -> MonoM Exp
 2 transformExp (Apply e1 e2 d tp loc) =
 3   case (e1, e2) of
 4   (..)
 5   (Var v _ _, TupLit [dest, op, ne, buckets, img] _)
 6     | intrinsic "gen_reduce" v ->
 7       transformExp $ GenReduce dest op ne buckets img loc
 8   where intrinsic s (QualName _ v) =
 9           baseTag v <= maxIntrinsicTag &&
10           baseName v == nameFromString s
11 (..)
12 transformExp (GenReduce e1 e2 e3 e4 e5 loc) =
13   GenReduce
14     <$> transformExp e1 -- histogram
15     <*> transformExp e2 -- operator
16     <*> transformExp e3 -- neutral element
17     <*> transformExp e4 -- buckets
18     <*> transformExp e5 -- input image
19     <*> pure loc
```

Figure 19: `Futhark.Internalise.Monomorphise`.

returns a tuple inside the `DefM`-monad. We are not experts on this part of the compiler, thus the reader must make do with a high-level explanation of the first component of the tuple which are the most important to us. The first component, `Exp`, resembles the original input value but instead of possible lambda abstractions it has record expressions capturing the environment variables at the time of evaluation. At a later point the function will be evaluated using the values from the record expressions. The defunctionalisation process is called recursively on the arguments of `GenReduce`, ultimately producing a defunctionalised `GenReduce`-node. The program is now a well-typed, monomorphic program, without modules and without higher-order functions.

Internaliser

Last, but definitely not least, we have the internaliser. It receives the program produced by the defunctionaliser, which is in source abstract syntax, and translates it into core abstract syntax. After this last translation step in the front end, the compiler will perform various transformations and optimizations on the core abstract syntax tree.

Before we go into detail about the implementation of the internaliser, we take

```
 1  defuncExp :: Exp -> DefM (Exp, StaticVal)
 2  defuncExp e@(GenReduce hist op ne bfun img loc) = do
 3    hist' <- defuncExp' hist
 4    op' <- defuncSoacExp op
 5    ne' <- defuncExp' ne
 6    bfun' <- defuncSoacExp bfun
 7    img' <- defuncExp' img
 8    return (GenReduce hist' op' ne' bfun' img' loc,
 9            Dynamic $ typeOf e)
10  (..)
```

Figure 20: `Futhark.Internalise.Defunctionalise`.

a step back and consider the high-level design of the internal representation of the new construct. We do this in order to show the difference between source Futhark, which uses an array-of-structs representation, and core Futhark, which uses a structs-of-arrays representation. For this reason, we cannot simply map the type for the source language construct to the core language. Here we develop the core language representation such that we know the end goal of this translation.

In order to describe the internal representation – the end point of the internalisation process – we will make use of the following notation adapted from [HLO16]. Whenever α is some object we will write $\overline{\alpha}$ to denote a sequence of α's:

$$\overline{\alpha} = \alpha \cdots \alpha$$

and if we need to specify the length of the sequence we add a superscript:

$$\overline{\alpha}^k = \alpha_1, \alpha_2 \cdots \alpha_k$$

We can also prepend to a sequence:

$$\beta\ \overline{\alpha}^k = \beta, \alpha_1, \alpha_2 \cdots \alpha_k$$

If the term under the bar is variant we add a subscript to the term:

$$\overline{\alpha_i}^k = \alpha_1, \alpha_2 \cdots \alpha_k$$

Finally, if we want to write out a sequence we use braces, e.g.:

$$\overline{\alpha}^2 = \{\alpha_1, \alpha_2\}$$

Now, referring to the function declaration for our construct in Figure 17, the source language type is as follows:

$$*[\texttt{m}]\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha \to [\texttt{n}]\texttt{i32} \to [\texttt{n}]\alpha \to *[\texttt{m}]\alpha$$

But, as mentioned, this has some implications. In particular, a source array $[](\texttt{i32}, \texttt{i32})$ is represented in the core language as two separate arrays of type $[]\texttt{i32}$. Using the new notation we are able to capture this fact in the core langauge as follows:

$$\begin{aligned}
&\overline{*[m_i]\alpha_i}^{\sum_k l_i} \\
\to\ &\overline{\left(\overline{\alpha_i}^{l_i} \to \overline{\alpha_i}^{l_i} \to \overline{\alpha_i}^{l_i}\right)}^{k} \\
\to\ &\overline{\alpha_i}^{\sum_k l_i} \\
\to\ &\left(\overline{\beta_i}^{k} \to \overline{\left(\texttt{i32},\ \overline{\alpha_i}^{l_i}\right)}^{k}\right) \\
\to\ &\overline{*[n_i]\beta_i}^{k} \\
\to\ &\overline{*[m_i]\alpha_i}^{\sum_k l_i}
\end{aligned}$$

where the first argument is a sequence of destination arrays, the second is a sequence of combining operators, the third is a sequence neutral elements, the fourth is a *single* bucket function, and the fifth is a sequence of input arrays. This ultimately produces a sequence of arrays of the same types and lengths as the first argument.

This representation allows k different input arrays from the source language, where each source array is represented by l_i arrays in the core language. For example, $[\texttt{m}](\texttt{f32}, \texttt{i32})$ in the source language, corresponds to $k = 1$, $l_i = 2$, and $\{[\texttt{m}]\texttt{f32}, [\texttt{m}]\texttt{i32}\}$ in the core language.

To gain some intuition for this representation consider the following example using a source array of type $[](\texttt{f32}, \texttt{f32})$. The type of the source language function becomes:

$$\begin{aligned}
&* [m](\texttt{f32}, \texttt{f32}) \\
\to\ &((\texttt{f32}, \texttt{f32}) \to (\texttt{f32}, \texttt{f32}) \to (\texttt{f32}, \texttt{f32})) \\
\to\ &(\texttt{f32}, \texttt{f32}) \\
\to\ &[n]\texttt{i32} \\
\to\ &[n](\texttt{f32}, \texttt{f32}) \\
\to\ &*[m](\texttt{f32}, \texttt{f32})
\end{aligned}$$

```
 1 data SOAC lore =
 2   (..)
 3   | GenReduce SubExp [GenReduceOp lore] (LambdaT lore) [VName]
 4   (..)
 5
 6 data GenReduceOp lore =
 7   GenReduceOp { genReduceWidth :: SubExp
 8               , genReduceDest :: [VName]
 9               , genReduceNeutral :: [SubExp]
10               , genReduceOp :: LambdaT lore }
```

Figure 21: `Futhark.Representation.SOACS.SOAC`.

while the type of the core language expression becomes:

$$
\begin{aligned}
& \{\ *[m]\texttt{f32},\ *[m]\texttt{f32}\ \} \\
\rightarrow\ & \{\ (\texttt{f32} \rightarrow \texttt{f32} \rightarrow \texttt{f32}),\ (\texttt{f32} \rightarrow \texttt{f32} \rightarrow \texttt{f32})\ \} \\
\rightarrow\ & \{\ \texttt{f32},\ \texttt{f32}\ \} \\
\rightarrow\ & (\beta \rightarrow \{\ \texttt{i32},\ \texttt{f32},\ \texttt{f32}\ \}) \\
\rightarrow\ & *[n]\beta \\
\rightarrow\ & \{\ *[m]\texttt{f32},\ *[m]\texttt{f32}\ \}
\end{aligned}
$$

where β is the type of the array that produces the indices and values when the bucket function is applied to it.

With this definition in place, we re-enter the pipeline where we left: right before the translation from source abstract syntax to core abstract syntax.

Like the source abstract syntax was parameterized by type information, the core abstract syntax is parameterized by a representation, where each representation eases different optimizations. There are three main representations in the Futhark compiler; SOACS, Kernels, and ExplicitMemory. We will go into details for each representation when we describe our implementation in the middle stage.

The end goal for the internalisation process is the core abstract syntax parameterized by the SOACs representation, meaning that program still has second-order array combinators and nested parallelism. Thus, we want to extend the collection of SOACs with our new construct based on the core language type developed above.

The Haskell implementation of the core language type, `GenReduce`, is shown in Figure 21. Its first argument, `SubExp`, is the length of the possibly multiple input arrays represented by the last argument, `[VName]`. This means that we

end up requiring all input arrays to be of equal length, although the core language type allowed otherwise. We do this in order for Futhark to perform better optimizations.

Further, we see that its second argument is `[GenReduceOp lore]`, representing the notion of a histogram in the source language, i.e., it contains a list of variable names (line 8) corresponding to components of the source input array, along with a list of neutral elements (line 9) represented in the same fashion as just described, and a combining operator (line 10). The first `SubExp` (line 7) is used later to ensure that all the arrays in line 8 have equal length. (Again, this goes against the core language type, and we are also going against it for the same reason as before.)

Finally, we have `(LambdaT lore)`, which is the sole bucket function.

Now that we have established a mapping between the source language and the core language, and added our construct to the existing collection of SOACS, we are ready to proceed with the actual translation. The main function of the internaliser is `internaliseExp`, which is shown in Figure 22. It is tasked with translating the source abstract syntax constructor, `GenReduce`, into the core abstract constructor of the same name. As an indication in the compiler source code, we see that the function takes as its second argument an `E.Exp` and returns `[I.SubExp]`, such that `E` refers to the source (external) abstract syntax, and `I` refers to the core (internal) abstract syntax parameterized by the SOACs representation.

`internaliseExp` takes two inputs; the second argument, `E.Exp`, is an expression from the source abstract syntax (in our case `GenReduce`) that we are about to translate into core abstract syntax, and the first argument is a string, which we want to bind the result of the translation of `Exp` too.

Roughly, the code can be divided into three sections:

- In lines 8-13 the individual components of the function are internalised recursively. For example, when internalising the variable `hist` it may result in multiple core language arrays as described previously.

- Because Futhark is statically typed, types are size-dependent, and we are doing these manipulations after the source abstract syntax has been type checked we insert assertions. For example, the compiler should throw an error when the row type of the destination array and the type of the neutral element is not referring to the same size variable (lines 17-22). In other words, you may have two variables that you know should be of the same size, but if they do not refer to the same size variable in the compiler, then it will complain.

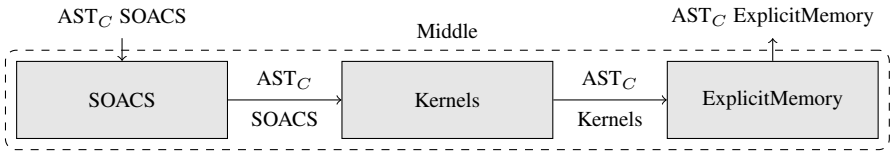
```
 1 (..)
 2 import Language.Futhark as E
 3 import Futhark.Representation.SOACS as I
 4 (..)
 5 internaliseExp :: String -> E.Exp -> InternaliseM [I.SubExp]
 6 internaliseExp desc
 7                (E.GenReduce hist op ne buckets img loc) = do
 8   ne' <- internaliseExp "gen_reduce_ne" ne
 9   hist' <- internaliseExpToVars "gen_reduce_hist" hist
10   buckets' <-
11     letExp "gen_reduce_buckets" . BasicOp . SubExp =<<
12     internaliseExp1 "gen_reduce_buckets" buckets
13   img' <- internaliseExpToVars "gen_reduce_img" img
14
15   -- reshape neutral element to have same size as the
16   -- destination array
17   ne_shp <- forM (zip ne' hist') $ \(n, h) -> do
18     rowtype <- I.stripArray 1 <$> lookupType h
19     ensureShape asserting
20       "Row␣shape␣of␣destination␣array␣does␣" ++
21       "not␣match␣shape␣of␣neutral␣element"
22       loc rowtype "gen_reduce_ne_right_shape" n
23
24   ne_ts <- mapM I.subExpType ne_shp
25   his_ts <- mapM lookupType hist'
26   op' <- internaliseFoldLambda
27          internaliseLambda op ne_ts his_ts
28
29   (..)
30
31   letTupExp' desc $ I.Op $
32     I.GenReduce w_img [GenReduceOp w_hist hist' ne_shp op']
33       (I.Lambda params body' rettype) $ buckets'' : img'
```

Figure 22: `Futhark.Internalise`.

- In lines 24-27 the combining operator, `op`, is internalised. Basically, the function `internaliseFoldLambda` is responsible for ensuring that the shapes of formal and actual arguments are correct, and that the return type of the function and the type of its body is the same.
- Finally, in lines lines 31-33, we return a variable that is let-bound to an internal `GenReduce`-expression comprising the internalised componenents.

This concludes the front end stage, where we have seen how the construct is exposed to the user as a polymorphic higher-order function, transformed into a monomorphic first-order function, and ultimately how it is translated into core abstract syntax.

5.2 Middle



After the internalization process, the program is now represented in core abstract syntax parameterized by the SOACs representation. As displayed in the figure above, the middle stage can be conceptually divided into three steps represented by the parameterization of the core abstract syntax tree that they are working on. Each step is comprised of multiple subpasses of which most are optimizing transformations such as fusion A high-level overview of the sub-passes are shown in Appendix B.1.

In the following, we will focus on enabling transformations, i.e., transformations that change the representation, and show how the new construct progresses through the pipeline.

Representation – SOACS

Since we have already seen how source abstract syntax is translated into core abstract syntax parameterized by the SOACs representation, we here show the corresponding type checker. Each representation extends a basic type checker and type checking is performed every time the core abstract syntax changes its parameterization. The type checkers are similar in structure and therefore we will not show the type checkers for the other representations.

Figure 23 displays the source code for the type checker for the SOACs representation. From the Haskell function declaration we see that it takes as input a

SOAC, in this case the new `GenReduce` node, and produces unit in the `TypeM`-monad. The monad contains a variable table and a function table, which the type checker has access to when it is running. This function, `typeCheckSOAC`, and other type checkers, then throw an error in the `TypeM`-monad when things are not type checking. These errors are indicated by, for example, the functions `TC.require` and `TC.bad`.

The type checker checks various properties, and we have emphasized the following:

- Line 4 ensures that the argument given as length for the input arrays is really an integer.
- Lines 7-23 are a larger code block that checks if the types of the combining operators corresponds to the type of their neutral element and to their destination arrays. For example, line 12 ensures that the arguments to the combining operator corresponds to the the type of the neutral elements, while lines 13-18 check that the return type of the operator equals the type of the neutral element.
- Finally, lines 32-40 check that the return type of the bucket function is an array of buckets, **`replicate`** `(`**`length`** `ops) (Prim int32)`, followed by the results from the combining operators, `nes_ts`.

With this, we conclude our description of the SOACs representation. As displayed in the small diagram below the title for this section, the core abstract syntax in the SOACs representation is now about to change representation to the Kernels representation.

Representation – Kernels

At this point in the pipeline, we introduce the notion of kernels, also indicated by name. This representation enables optimizations related to flattening and coalescing, but our implementation has only touched the enabling transformation.

The starting point is the `GenReduce`-node from the SOACs representation introduced in Figure 21, and the end point is a new node named `GroupGenReduce` (Figure 24), which captures the notion of in-kernel expressions. In other words, `GroupGenReduce` belongs to a group of expressions, such as `GroupScan` and `GroupReduce`, which can be said to be in the body of a kernel. This is similar to how `GenReduce` extended the collection of SOACs.

With the change of representation, we introduce the notion of kernels, which means that the program now supports flat-parallelism expressed via GPU kernels. This is the major first difference between the parallel and sequential pipeline.

```
typeCheckSOAC :: TC.Checkable lore =>
                 SOAC (Aliases lore) -> TC.TypeM lore ()
typeCheckSOAC (GenReduce len ops bucket_fun imgs) = do
  TC.require [Prim int32] len

  -- Check the operators.
  forM_ ops $ \(GenReduceOp dest_w dests nes op) -> do
    nes' <- mapM TC.checkArg nes
    TC.require [Prim int32] dest_w

    -- Operator type must match the type of neutral elements.
    TC.checkLambda op $ map TC.noArgAliases $ nes' ++ nes'
    let nes_t = map TC.argType nes'
    unless (nes_t == lambdaReturnType op) $
      TC.bad $ TC.TypeError $ "Operator␣has␣return␣type␣" ++
      prettyTuple (lambdaReturnType op) ++
      "␣but␣neutral␣element␣has␣type␣" ++
      prettyTuple nes_t

    -- Arrays must have proper type.
    forM_ (zip nes_t dests) $ \(t, dest) -> do
      TC.requireI [t `arrayOfRow` dest_w] dest
      TC.consume =<< TC.lookupAliases dest

  -- Types of input arrays must equal parameter types
  -- for bucket function.
  img' <- TC.checkSOACArrayArgs len imgs
  TC.checkLambda bucket_fun img'

  -- Return type of bucket function must be an index for
  -- each operation followed by the values to write.
  nes_ts <- concat <$>
    mapM (mapM subExpType . genReduceNeutral) ops
  let bucket_ret_t =
    replicate (length ops) (Prim int32) ++ nes_ts
  unless (bucket_ret_t == lambdaReturnType bucket_fun) $
    TC.bad $ TC.TypeError $
    "Bucket␣function␣has␣return␣type␣" ++
    prettyTuple (lambdaReturnType bucket_fun) ++
    "␣but␣should␣have␣type␣" ++ prettyTuple bucket_ret_t
(..)
```

Figure 23: `Futhark.Representation.SOACS.SOAC`.

```
data KernelExp lore =
  (...)
  | GroupGenReduce [SubExp] [VName] (LambdaT lore)
                   [SubExp] [SubExp] VName
  (...)
```

Figure 24: `Futhark.Representation.Kernels.KernelExp`.

```
1 transformStm :: KernelPath -> Stm -> DistribM KernelsStms
2 transformStm path (Let orig_pat (StmAux cs _)
3               (Op (GenReduce w ops bucket_fun imgs))) = do
4   bfun' <- Kernelise.transformLambda bucket_fun
5   genReduceKernel path [] orig_pat [] [] cs w ops
6                   bfun' imgs
```

Figure 25: `Futhark.Pass.ExtractKernels`.

On a high level, we are transforming a program using second-order array combinators to a program using explicit kernels, by applying a kernel extraction transformation. The transformation will attempt to rearrange construct in order to make more parallelism available, e.g., via loop-distribution.

In the process of translating a `GenReduce`-node to a `GroupGenReduce`-node, we will use three functions, namely, `transformStm`, `genReduceKernel`, and `blockedGenReduce`. Together they generate code for, among other things, the final segmented reduction that combines the subhistograms, the loop that ensures coalesced reads, the heuristic, and for computing indices for reading and writing memory.

The entry point for the translation is `transformStm`, which is shown in Figure 25. It takes two inputs, but only the second, `Stm`, is important to us. A `Stm` is basically a let-bound variable. Recall, that we bound the result of processing the `GenReduce`-node in the function `internaliseExp` in Figure 22.

`transformStm` is mainly a wrapper for `genReduceKernel`, but first it extracts the statements in the body of the bucket function (line 4).

This brings us to `genReduceKernel` which is shown in Figure 26. It takes as input a lot of arguments, but for simplicity, we will focus on the last four, which corresponds to the arguments to the `GenReduce`-node in the SOACS representation (Figure 21). The functions overall purpose is twofold: 1) it adds the statements produced by the call to `blockedGenReduce` (line 9), which we will return to, and 2) it generates code for choosing at runtime if we can avoid the

final segmented reduction.

Besides the call to `blockedGenReduce` in line 9, we will empasize three important sections:

- Lines 13 and 15 chunks the variable names referring to arrays in the destination histogram and the variable names we want to bind them to. This is due to the fact that one histogram may comprise multiple arrays in Haskell caused by the tuples-of-arrays representation in the Futhark compiler.
- Lines 22-23 computes the number of subhistograms we are using for each histogram computation. This is not part of the heuristic, but is merely a simple trick to possibly avoid performing the final segmented reduction if we have only one subhistogram.
- Lines 26-29 then checks if the number of subhistograms are equal to one (still on a per histogram basis), which are used in lines 32-36 to generate an if-statement that chooses between two code generations at runtime; `body_with_segred` contains code for the segmented reduction case, while `body_with_reshape` contains code for the single subhistogram case,

Above we mentioned, that `genReduceKernel` is responsible for adding the statements produced by `blockedGenReduce`, but we did not specify what statements. As an example, it is tasked with creating the loop that ensures coalesced reads, but also to allocate the locks array.

`blockedGenReduce`, which is shown in Figures 27 and 28, is a rather large function and a lot of things are happening, but we will empasize four important sections. The first two are in the shown in Figure 27:

- Lines 5-15 checks the number of subhistograms. If we have one large histogram we reuse the original array as the destination. If we have many subhistograms we copy the values from the original array into the destination array. Finally, we insert an if-statement to choose at runtime.
- Lines 18-21 are responsible for allocating the locks array, which is an integer array with length equal to the number of buckets times the number of subhistograms, and with all elements initialized to zero. To allocate the array we insert a `Replicate`-node.

  Note, that the locks array is allocated regardless of the operator, although it is only the **`atomicExch`** strategy that needs this array. This is because we select the strategy depending on the operator late in the pipeline, as we will see later. We discuss the implications of this design in Section 7.1 when we consider the subhistogramming strategy in shared memory.

```
genReduceKernel path nests orig_pat ispace
                inputs cs genred_w ops lam arrs = do
  ops' <- forM ops $ \(GenReduceOp num_bins dests nes op) ->
    GenReduceOp num_bins dests nes <$>
                Kernelise.transformLambda op
  (..)
  runBinder_ $ do
    (histos, k_stms) <-
       blockedGenReduce genred_w ispace inputs' ops' lam arrs

    addStms $ fmap (certify cs) k_stms

    let histos' = chunks (map (length . genReduceDest) ops')
                      histos
        pes = chunks (map (length . genReduceDest) ops') $
              patternElements orig_pat

    mapM_ combineIntermediateResults (zip3 pes ops histos')
  where (..)
        combineIntermediateResults
          (pes, GenReduceOp num_bins _ nes op, histos) = do
          num_histos <- arraysSize depth <$>
                        mapM lookupType histos

          -- Avoid the segmented reduction if num_histos is 1.
          num_histos_is_one <-
            letSubExp "num_histos_is_one" $
            BasicOp $ CmpOp (CmpEq int32) num_histos $
                                          intConst Int32 1

          (..)
          letBindNames (map patElemName pes) $
            If num_histos_is_one
              body_with_reshape body_with_segred $
              IfAttr (staticShapes $ map patElemType pes)
                IfNormal
```

Figure 26: `Futhark.Pass.ExtractKernels`.

```
 1 blockedGenReduce arr_w segments inputs ops lam arrs =
 2   runBinder $ do
 3   (..)
 4   -- Initialize sub-histograms.
 5   sub_histos <- forM (zip ops num_histos) $
 6       \(GenReduceOp w dests nes _, num_histos') -> do
 7     -- If num_histos' is 1: reuse the original destination
 8     let num_histos_is_one =
 9           BasicOp $ CmpOp (CmpEq int32) num_histos' $
10                                         intConst Int32 1
11         reuse_dest = (..)
12         make_subhistograms = (..)
13     letTupExp "histo_dests" =<<
14       eIf (pure num_histos_is_one) reuse_dest
15                                   make_subhistograms
16   (..)
17   lock_arrs <- forM (zip ops num_histos) $
18       \(GenReduceOp w _ _ _, num_histos') ->
19     letExp "locks_arr" $ BasicOp $
20     Replicate (Shape $ segment_sizes ++ [num_histos', w])
21               (intConst Int32 0)
22   (..)
```

Figure 27: `Futhark.Pass.ExtractKernels.BlockedKernel`.

The third and fourth sections are shown in Figure 28:

- Lines 2-25 are responsible for creating the statements that goes into the kernel, `kstms`, and for computing the location to which the results of the kernel are written, `kres`. In line 27 the two are bound in a kernel body, and used in lines 28-30 to create a kernel expression which is bound to a variable named `histograms`.
- Lines 4-22 are responsible for creating the body of the coalescing loop. Besides what is shown in the code snippet, the loop body also computes the offset into the input array is computed, the bucket function is applied, and it is checked whether the bucket is in-bounds.

  What *is* shown is that we compute which subhistogram we are working on, and that we create a `GroupGenReduce` vale constructor from Figure 24. The constructor contains all the necessary information for the intermediate code generator to create intermediate code for the construct, e.g., the sub-

```
1    (..)
2    (kres, kstms) <- runBinder $ (..)
3      (..)
4      loop_body <- runBodyBinder $ (..)
5        (..)
6        lam_res <- letTupExp "bucket_fun_res" =<<
7                     eIf in_bounds in_bounds_branch
8                     not_in_bounds_branch
9        (..)
10       ops_res <- forM (..) $
11         \(GenReduceOp dest_w _ _ comb_op, subhistos,
12           bucket, vs', lock_arrs', num_histos') -> do
13
14           -- Compute subhistogram index for each thread.
15           subhisto_ind <- (..)
16           fmap (map Var) $ letTupExp "genreduce_res" $ Op $
17             GroupGenReduce (segment_sizes ++ [num_histos', dest_w])
18             subhistos comb_op (map Var segment_is ++
19                                  [subhisto_ind, bucket])
20                                vs' lock_arrs'
21
22       return $ resultBody $ concat ops_res
23
24     result <- letTupExp "result" $ DoLoop [] merge form loop_body
25     return $ map KernelInPlaceReturn result
26
27   let kbody = KernelBody () kstms kres
28   letTupExp "histograms" $ Op $ Kernel
29                                 (KernelDebugHints "gen_reduce" [])
30                                 kspace dest_ts kbody
```

Figure 28: `Futhark.Pass.ExtractKernels.BlockedKernel` (cont'd).

histograms, `sub_histos`, the operator, `comb_op`, the bucket, `bucket`, the values, `vs'`, and the locks array, `lock_arrs'`.

In line 24 the statements for creating the components just described are used as the body for a for-loop, which are then returned and used as the statements described in the previous bullet.

With this, we conclude our presentation of the Kernel representation stage. Here we have seen how the `GenReduce`-node from the SOACs representation

```
 1 handleKernel (Kernel desc space kernel_ts kbody) =
 2   subAllocM handleKernelExp True $
 3   (..)
 4   where handleKernelExp (GroupGenReduce w dests op
 5                          bucket vs locks) = do
 6         (..)
 7           op' <- allocInLambda (x_params'<>y_params')
 8                                (lambdaBody op)
 9                                (lambdaReturnType op)
10           return $ Inner $
11                    GroupGenReduce w dests op' bucket vs locks
```

Figure 29: `Futhark.Pass.ExplicitAllocations`.

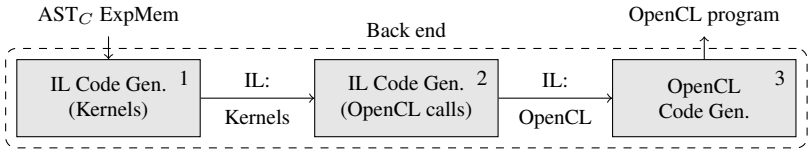
has been translated into the `GroupGenReduce`-node in the Kernels representation, and with this translation how the program now has a notion of GPU-oriented kernels.

Representation – ExplicitMemory

Finally, we arrive at the last representation, namely, the ExplicitMemory representation. A lot of low-level and intricate bookkeeping happens within this representation and our implementation has not dealt much with this pass. Every array expression in Futhark resides in a memory block which was explicity allocated in the program. This pass translates the core abstract syntax using the Kernels representation, which has no notion of memory annotations, into ExplicitMemory representation. During this translation statements that correspond to arrays are assigned to memory blocks. Thus, our implementation simply utilizes the existing infrastructure.

We are, admittedly, not experts in this complicated stage, but for completeness Figure 29 displays the main function, `handleKernel`, responsible for handling the new construct. At this point, the construct is represented by the `GroupGenReduce`-node and `handleKernel` wraps it in an `Inner`-node, after having checked if any arrays need to be assigned to memory blocks inside the combining operator (lines 7-9).

5.3 Back End



The high-level idea of the back end is straightforward. As the figure above shows, the back end receives a program in abstract core syntax parameterized by the ExplicitMemory representation and translates it into an imperative intermediate language, before creating an OpenCL program.

Currently, Futhark does not have support for generating atomic functions, and since we aim at using them, we need to add support for atomic functions. Neither the source nor the core abstract syntax are to be executed, at least not in a parallel setting, thus they need not have a notion of atomic functions. Thus, we extend the intermediate language, along with the existing intermediate-code generator to be able to generate intermediate code containing atomic functions.

The main modifications were done in box one in the diagram below the section title, which pertains to extending the intermediate language and the corresponding intermediate-code generator. Because we extend the intermediate language we also need to extend the OpenCL code generator, which pertains to box two. For the remainder of the pipeline we rely on the existing infrastructure, and box three is merely included for completeness.

To assist both our understanding and the extension of the back end we created call-graphs for the programs `futhark-opencl` and `futhark-c` which are displayed in Appendix B.2.

This section is split into two subsections, one for each of the first two boxes in the diagram above, and we start with the leftmost.

5.3.1 IL Code Generation – Kernels

At this point, the program is in core abstract syntax parameterized by the ExplicitMemory representation, and is going to be translated into intermediate language. The latter contains no notion of atomic functions and for that reason it needs to be extended.

The first step is to extend the category of expressions that can go into kernels (`Futhark.CodeGen.ImpCode.Kernels`) with a notion of atomic functions:

```
data KernelOp = GetGroupId VName Int
              | GetLocalId VName Int
              (..)
              | Atomic AtomicOp
              | Barrier
              | MemFence
              deriving (Show)

data AtomicOp = AtomicAdd VName VName (Count Bytes) Exp
              | AtomicCmpXchg VName VName (Count Bytes)
                              Exp Exp
              | AtomicXchg VName VName (Count Bytes) Exp
              (..)
              deriving (Show)
```

i.e., we extend the data type `KernelOp` with the value constructor `Atomic` which takes one argument, namely, a value constructor of the type `AtomicOp` which we also added. `KernelOp` also contains a memory fence constructor.

`AtomicOp` contains constructors for all atomic functions available in OpenCL, but most importantly `atomic_add`, `atomic_cmpxchg`, and `atomic_xchg`, but also functions such as `atomic_max`. Later we will se how the atomic functions is used for implementing the locking strategies presented earlier.

With the notion of atomic functions in the intermediate language we extend the intermediate-code generator, enabling it to translate the `GroupGenReduce` expression from core abstract syntax to intermediate imperative code. This part constitutes an important piece of the implementation as it is directly responsible for generating intermediate code for the strategies shown in Section 4.1.

The function responsible for the translation is `compileExp`. It has two patterns for the `GroupGenReduce` constructor: The first is for situations in which the strategies **`atomicAdd`** or **`atomicCAS`** can be used, displayed in Figures 30 and 31, respectively. The other is a fallback case, shown in Figure 32, which is for situations where the **`atomicExch`** strategy has to be used.

In both cases `compileKernelExp` only uses the last argument, namely, the `GroupGenReduce`-node created in the Kernels stage. It produces statements that go into kernels via the `ImpM`-monad (`InKernelGen()` is simply a type synonym), where the environment in the monad is updated with the produced intermediate code.

Starting with the **`atomicAdd`**-case in Figure 30, the case-expression in line 8 recognizes whether the operator has a corresponding function in OpenCL. If it does, this function is returned, and the value that we want to apply the function to is translated (line 10). Finally, we generate intermediate code for an if-statement that applies the atomic function if the index is in-bounds (line 12) and performs

```
 1 compileKernelExp :: KernelConstants -> ImpGen.Destination
 2                   -> KernelExp InKernel
 3                   -> InKernelGen ()
 4 compileKernelExp _ _ (GroupGenReduce w [a] op bucket [v] _)
 5   | [Prim t] <- lambdaReturnType op,
 6     primBitSize t == 32 = do
 7   (..)
 8   case opHasAtomicSupport old arr' bucket_offset op of
 9     Just f -> do
10       val' <- ImpGen.compileSubExp v
11       ImpGen.emit $
12         Imp.If (indexInBounds bucket' w')
13         (Imp.Op $ f val')
14         Imp.Skip
15     Nothing -> do
16    (..)
```

Figure 30: `Futhark.CodeGen.ImpGen.Kernels`. Generating intermediate code for the **`atomicAdd`** case.

a no-operation if it is out-of-bounds (line 14).

If the operator does not have a dedicated atomic function it can still be implemented using the **`atomicCAS`** strategy shown in Section 4.1.1. Since the intermediate language does not have do-while-loops we implemented the strategy using a while-loop:

```
loop = true;
while(loop) {
  assumed = old;
  old = atomicCAS(histo[bucket], assumed, assumed + val)
  if(assumed == old) {
    loop = false;
  }
}
```

The implementation is displayed in Figure 31 and can roughly be divided into three sections:

- In lines 11-12 we assign `assumed = old`. Furthermore, we manually copy the to-be written value, `val`, and the current value, `assumed`, to the parameters of the lambda-body of the operator (`bind_acc_param` and

`bind_arr_param`). The operator binds the result of the computation to the same variable as `bind_acc_param`.

- Lines 14-20 corresponds to the atomic function, where `acc_p` is the variable bound in `bind_acc_param`.

  In addition to this, one might notice the functions `toBits` and `fromBits`. This trick makes it possible to use the atomicCAS strategy on 32-bit floats as well, i.e., before writing the value with atomicCAS the float is converted to bits which are then written as a 32-bit integer. This is an important part of the flexibility of the implementation, as many high-performance problems are using floats. (One might note, that floating point addition is not associative, but people are doing it anyway.)

- Finally, lines 22-29 corresponds to the if-statement, responsible for breaking out of the while-loop when the value was correctly updated.

Finally, we have the fallback case using the **`atomicExch`** strategy, i.e., if the operator does not have atomic support and cannot be implemented as a binary operator on one memory location. The code generation is displayed in Figure 32 and can roughly be divided into four sections:

- Lines 6-7 compute the correct index into the locks array.

- In lines 10-19 we declare a handful of code generations up front in order to express the code generation for the while-loop more compactly. Most importantly, this includes the atomic function trying to acquire the lock, `try_acquire_lock`, by setting the integer in the locks array at the index corresponding to the bucket. In this way, we set only one lock even though the histogram may consist of multiple arrays in the source language.

- Next, we have the while-loop, which is entered if the index is in-bounds, and keep looping until the lock is acquired and the value is set. In line 24 we make an attempt to acquire the lock and in the following line we check if we succeeded. If we got the lock, we perform a handful of operations: Like for the previous cases we bind parameters to the formal arguments of the operator. Next, we update the possible multiple arrays with the new values (`update_arrs`), release the lock (`release_lock`), and break out of the loop (`break_loop`).

- Finally, we have the memory fence in line 32.

With this, we conclude the intermediate-code generation. We we have now seen how the claimed flexibility for user-defined functions and support for floats

```
compileKernelExp _ _ (GroupGenReduce w [a] op bucket [v] _)
  | [Prim t] <- lambdaReturnType op,
    primBitSize t == 32 = do
  (..)
  case opHasAtomicSupport old arr' bucket_offset op of
      Just f -> do
      (..)
      Nothing -> do
          (..)
          ImpGen.emit $ Imp.While (Imp.var run_loop int32)
            (Imp.SetScalar assumed (Imp.var old t) <>
             bind_acc_param <> bind_arr_param <> op_body
             <>
             (Imp.Op $
                 Imp.Atomic $
                   Imp.AtomicCmpXchg old_bits arr' bucket_offset
                     (toBits (Imp.var assumed int32))
                     (toBits (Imp.var (paramName acc_p) int32)))
             <>
             Imp.SetScalar old (fromBits (Imp.var old_bits int32))
             <>
              Imp.If
                (Imp.CmpOpExp
                  (CmpEq int32) (toBits $ Imp.var assumed t)
                  (Imp.var old_bits int32))
                -- True branch:
                (Imp.SetScalar run_loop 0)
                -- False branch:
                Imp.Skip
            )
```

Figure 31: Generating intermediate code for the **`atomicCAS`** case. (`Futhark.CodeGen.ImpGen.Kernels`.)

```
 1  (..)
 2  compileKernelExp _ _
 3    (GroupGenReduce w arrs op bucket values locks) = do
 4    (..)
 5    -- Correctly index into locks.
 6    (locks', _locks_space, locks_offset) <-
 7      ImpGen.fullyIndexArray locks bucket'
 8
 9    (..)
10    ImpGen.declaringLParams (lambdaParams op) $ do
11      let try_acquire_lock =
12            Imp.Op $ Imp.Atomic $
13            Imp.AtomicXchg old locks' locks_offset 1
14          lock_acquired =
15            Imp.CmpOpExp (CmpEq int32) (Imp.var old int32) 0
16          loop_cond =
17            Imp.CmpOpExp (CmpEq int32) (Imp.var loop_done int32) 0
18          break_loop =
19            Imp.SetScalar loop_done 1
20
21      (..)
22      -- While-loop: Try to insert your value
23      ImpGen.emit $ Imp.While loop_cond
24        (try_acquire_lock <>
25          Imp.If lock_acquired
26           -- True branch
27           (bind_acc_params <> bind_arr_params <> op_body <>
28            update_arrs <> release_lock <> break_loop)
29           -- False branch
30           Imp.Skip
31           <>
32          Imp.Op Imp.MemFence
33        )
```

Figure 32: Generating intermediate code for the **`atomicExch`** case. (`Futhark.CodeGen.ImpGen.Kernels`.)

is actually implemented in the compiler. This leads us to the final modification in order to support the new construct.

5.3.2 IL Code Generation – OpenCL Calls

In this phase, the intermediate program is augmented with OpenCL kernels, which are merely strings containing the OpenCL kernel source code. Because we extended the intermediate language with a notion of atomic functions we need to be able to translate these intermediate language constructs into actual OpenCL code. More specifically, we need to add a case recognizing each of the value constructors, e.g., `AtomiAdd` or `AtomicCmpXchg`, to the function generating OpenCL code:

```
1  (..)
2  atomicOps (AtomicAdd old arr ind val) = do
3    ind' <- GenericC.compileExp $ innerExp ind
4    val' <- GenericC.compileExp val
5    GenericC.stm [C.cstm|$id:old =
6  atomic_add((volatile __global int *)&$id:arr[$exp:ind'],
7             $exp:val');|]
8    (..)
```

In lines 3 and 4 we recursively translate the index and value, respectively, which are then used to construct an expression containing the atomic function in lines 5-7. The function, and much of the back end in general, is based on Template Haskell for generating the target program AST. Basically, everything in between the two pipes is OpenCL code and the `$id` and `$exp` identifiers to capture a variable from the Haskell environment and use it in the OpenCL code. In this way we are able to translate the atomic functions to OpenCL code while inserting the bucket and the value. The translation of the rest of the function is performed similarly, and thus we have omitted it.

With this, we conclude our description of the implementation. We have seen how the construct was exposed to the user via the library function **`reduce_by_index`** and how it was translated into core abstract syntax. From here, we saw its path through the middle stage, progressing through the representations. Along the way, we made the necessary extensions to support the new construct. Ultimately, we extended the intermediate language in the back end with a notion of atomic operations and added support for translating these extensions into actual OpenCL code.

6 Validation and Benchmarks

In order to evaluate our new construct, we consider two different properties. The first property is whether our implementation yields efficient programs and, importantly, if we obtain speedups as claimed. The second property is whether our implementation increases readability of programs and is intuitive to use.

The chapter is split into two parts. First, we will look at the performance of the current strategies for histogram computation presented in Chapter 3, along with a reference implementation in Thrust. Next, we will rewrite two existing Futhark programs from the Parboil and Rodinia benchmark suites. In addition, we port a benchmark program from Parboil. Both sections evaluate both properties mentioned above.

Code for reproducing all results presented in this chapter is available here:

`https://github.com/lolkat2k/masters-benchmarks/`

and were run on a system consisting of an Intel Xeon E5-2650 CPU and a NVIDIA GTX 780Ti GPU, which is the same as in Chapter 4.

6.1 Micro-benchmarks

This section contains a small experiment and its purpose is twofold: First, we want to measure if our implementation effort pays off in performance, i.e., is the performance of the new construct better than the solutions described in Chapter 3. Second, we want to compare the performance to the outside world, in this case, a reference implementation in Thrust.

In this experiment, all programs compute a simple histogram, and each program takes as input only the number of buckets and an array of indices, where no index is out of bounds. Each program was run five times and the average was used to report either the runtime or compute the speedup. All datasets contains 20 million indices.

The results are displayed in Figure 34, where **REF** is a sequential C program used as baseline, **FV1** is the sort-reduce composition, **FV2** is the **`stream_red`**

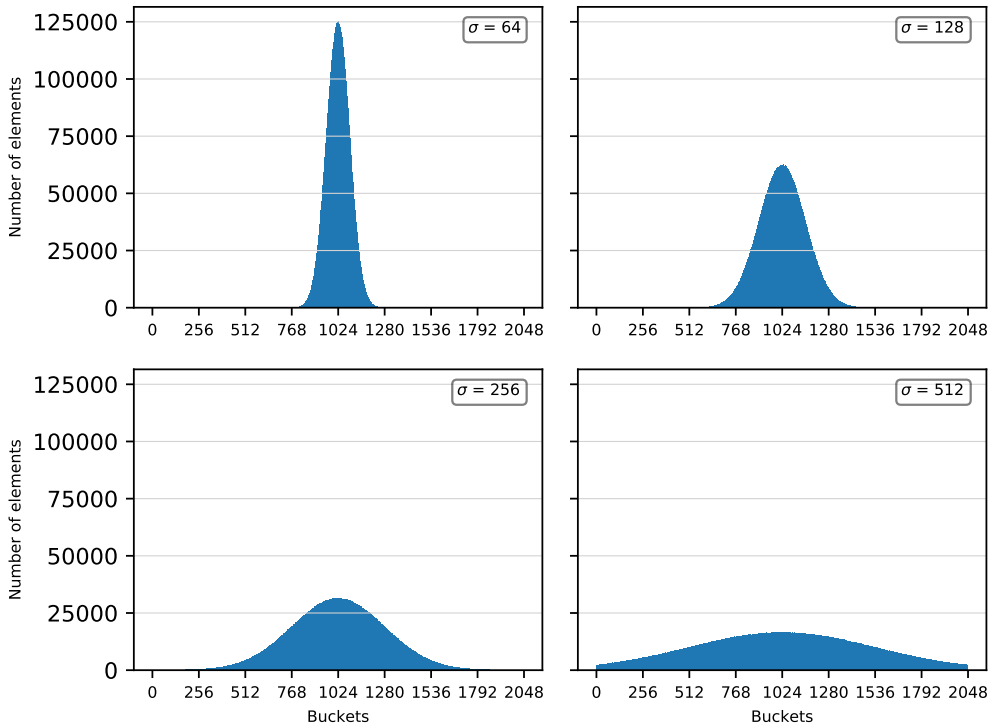


Figure 33: Twenty million elements from a truncated normal distribution with lower limit 0, upper limit 2048, $\mu = 1024$, and varying standard deviations, corresponding to datasets **D5**-**D8**, respectively.

program, **FV3** is the new **`reduce_by_index`** using the **`atomicAdd`**-based implementation in global memory, and **Thrust** is a histogram computation implemented in Thrust (see Chapter 3).

The programs were evaluated on twelve datasets grouped in four categories:

D1-D4: The indices are uniformly distributed, and the number of buckets is 16, 256, 4096, 65536, respectively.

D5-D8: The number of buckets is fixed at 2048, and indices are drawn from a truncated normal distribution with mean 1024, and standard deviation 64, 128, 256, and 512, respectively. Figure 33 illustrates the distribution of indices.

D9-D12: The number of buckets are 16, 256, 4096, and 65536 (the same sizes as **D1**-**D4**), but for each dataset all indices are hitting only one bucket, i.e., all indices in the dataset are equal to bucket size divided by two.

	REF (ms)	**FV1**(×)	**FV2**(×)	**FV3**(×)	**Thrust**(×)
D1	13ms	0.26×	2.40×	12.88×	0.90×
D2	13ms	0.15×	0.63×	8.02×	0.85×
D3	13ms	0.10×	0.32×	5.31×	0.80×
D4	41ms	0.25×	-	16.11×	2.48×
D5	13ms	0.11×	0.44×	5.86×	0.82×
D6	13ms	0.11×	0.43×	4.83×	0.81×
D7	13ms	0.11×	0.42×	6.05×	0.81×
D8	13ms	0.11×	0.42×	5.52×	0.80×
D9	36ms	0.71×	13.49×	17.63×	2.51×
D10	36ms	0.41×	9.66×	12.66×	2.50×
D11	36ms	0.28×	1.70×	6.24×	2.50×
D12	36ms	0.22×	-	1.62×	2.50×

Figure 34: Speedups of three Futhark programs (**FV1**-**FV3**) and a Thrust program (**Thrust**) compared to a sequential C program (**REF**) on twelve datasets (**D1**-**D12**). Column **REF** acts as baseline and displays runtime in milliseconds; **FV1**-**FV3** and **Thrust** display speedups.

We notice that the results confirm our intuition about the new construct. For example, if we compare datasets **D4**, where indices are uniformly distributed, and **D12**, where all indices hit the same bucket, we see a speedup of $16.11\times$ for the former and only $1.62\times$ for the latter, even though the histogram are of the same size (65536). In both cases, all threads will cooperate on one histogram, but **D4** corresponds to the best case in terms for collisions, while **D12** corresponds to the worst case. This pattern is clear if we look only at datasets **D9**-**D12**: The larger the histogram the fewer subhistograms will be allocated, resulting in a higher frequency of collisions.

Comparing within datasets **D1**-**D4** we see an odd pattern, namely, that the speedup decreases with histogram size, but for the largest histogram size (**D4**) we suddenly see a huge speedup. This is not due to properties of `reduce_by_index` as it takes around $100\mu s$ longer to run **D4** than **D3**, as we expected. Instead, the speedup is due to the impact on caching in the sequential reference solution, caused by the random writes in a large array.

Looking at datasets **D5**-**D8** we see that the speedup fluctuates around five for all values of sigma. At first glance, we would have expected the speedup to increase with decreasing values of sigma. (See Figure 33.) But if we take into account two factors this might not be as surprising: First, the small number of

buckets (2048) allows for many subhistograms which reduces the frequency of collisions. This by itself is not enough as more collisions will occur with increasing values of sigma. But the second factor, and probably the most important, is that we are utilizing the atomic addition function, which is very effective. Had we instead used an operator that does not have an atomic counterpart, we would probably have seen a decrease in speedup with increasing values of sigma.

If we compare across dataset groups **D1**-**D4** and **D9**-**12** one would intuitively expect the speedup to be smaller for the latter group than the first. Instead, we see the opposite. We conjecture that this is primarily caused by a high cache hit rate for the latter. This leads us to consider the dataset group **D5**-**D8**: If less variance in the dataset causes better cache behavior, then **D5** should result in a higher speedup than **D8**. We guess that the existing variance in the low sigma datasets are still too much for improved cache behavior to have an impact on performance, but we are not at all sure.

Furthermore, we see that the sort-reduce composition (**FV1**) is outperformed for all datasets, although stable for datasets **D5**-**D8**. Both properties are as expected. The `stream_red` (**FV2**) operator performs well on small datasets but fails to handle large histograms (**D4** and **D12**).

Finally, we note that the Thrust reference implementation performs poorly, but we will not put any emphasis on this. We are *not* Thrust experts, and so our implementation may be very poor. But the stable results support the implementation intuition as indices are sorted before being reduced. The reduction phase appears to be a segmented reduction. Furthermore, we wave a suspicion that some memory copying is performed behind the curtains due to our approach.

In total, the results confirm the design intuition: It performs well on small to medium histograms with uniformly distributed data; less well when data is not uniformly distributed; poor for large histograms where only a few, far distant buckets are hit.

6.2 Established Benchmarks

In addition to the datasets crafted by us, we will test `reduce_by_index` on programs from the wild: We rewrite two existing benchmark programs already implemented in Futhark, namely, TPACF from Parboil and k-means from Rodinia, and we port the Histo benchmark from Parboil. In the following, we will for each benchmark describe and analyze the results, starting with the two programs from Parboil.

6.2.1 Parboil

TPACF

Basically, the interesting part for in this context of the TPACF Futhark program is a map over a sequential histogram computation, where all subhistograms are combined. This means, that we should be able to simply flatten all indices and perform the histogram computation in one go.

From analyzing the program and the datasets we see that the histograms are small, around 20 bins in the interesting function, on a large number of elements.

Figure 35(a) displays the part of the program that we have modified. Starting in line 6 we see that it maps a function `one_value`, which body is a sequential loop computing a histogram, over some arrays. This produces an array of histograms which are piped into a function `sumBins` that combines the histograms in exactly the same way as the Futhark reduction program from Section 4.3.1. We recognize that this is one of the current strategies shown in Section 3.2, for computing histograms in Futhark, and so it should be possible to rewrite it to use **`reduce_by_index`**.

The rewritten program is shown in Figure 35(b), and the first thing we do is to flatten the indices inside the mapped function, such that they can be given to **`reduce_by_index`** as an argument (line 4). Then we apply **`reduce_by_index`** to the indices, effectively creating a histogram that may exploit the inner parallelism.

In addition to this, we tried compiling the rewritten program with the incremental flattening feature in Futhark, which basically generates different code versions to choose from at runtime. Each version extracts different amounts of inner parallelism, and the one that extracts the least but still enough parallelism is chosen. For the TPACF Futhark program this means that the inner **`reduce_by_index`** is sequentialised, effectively reverting our rewrite to look like the original program.

The above programs were tested on three datasets from the Parboil benchmark suite. The results are displayed in Figure 36 where **REF** is the OpenCL NVIDIA implementation from the Parboil benchmark suite, which is used as baseline and displaying runtime in milliseconds. **FV1** is the current implementation in Futhark, **FV2** is the rewritten implementation, and **FV3** is as **FV2** but compiled using the incremental flattening feature in Futhark. Programs **FV1**-**FV3** are compared to **REF** and display speedups.

From the results, we see our rewritten program takes more then double the time than the reference implementation, while the current solution is 25% slower. Compiling with the incremental flattening strategy, which as mentioned sequentialised the inner **`reduce_by_index`**, obtains roughly the same runtime as the

```
1 let one_value (xOuter, yOuter, zOuter) index =
2   loop dBins = replicate numBins2 0i32
3     for j in index+1..<numD do
4       -- compute index (sequential)
5     in unsafe dBins with [index] <- dBins[index] + 1i32
6 in map2 one_value data1 (iota numD) |> sumBins
```

(a) Original part.

```
1 let one_value (xOuter, yOuter, zOuter) =
2   let myFun (xInner, yInner, zInner) =
3     -- compute index (sequential)
4   let indices = map myFun data2
5   in reduce_by_index (replicate numBins2 0)
6                      (+) 0 indices
7                      (replicate num2 1)
8 in map one_value data1 |> sumBins
```

(b) Rewritten part.

Figure 35: Rewriting the TPACF Futhark program.

	D1	**D2**	**D3**
REF (ms)	8ms	435ms	2668ms
FV1 (×)	0.62×	0.77×	0.76×
FV2 (×)	0.45×	0.47×	0.43×
FV3 (×)	0.61×	0.77×	0.76×

Figure 36: TPACF benchmark. Speedup of three Futhark programs (**FV1**-**FV3**) compared to an OpenCL baseline implementation (**REF**) from the Parboil benchmark suite, compared on three datasets (**D1**-**D3**) also from the Parboil benchmark suite. Row **REF** is the baseline and displays runtime in miliseconds; **FV1**-**FV3** display speedups.

	D1
REF (μs)	297μs
FUT ($\times$)	1.7$\times$

Figure 37: Histo benchmark. Speedup of the Futhark implementation (**FUT**) compared to an OpenCL baseline implementation (**REF**) from the Parboil benchmark suite, compared on a single dataset (**D1**) also from the Parboil benchmark suite. Row **REF** is the baseline and displays runtime in microseconds; **FUT** displays speedup.

current solution. This indicates and leads us to believe, that the current solution is the optimal in Futhark, at least for now.

In addition to the Futhark programs presented above, we also tried a strategy that flattened all indices, such that only one **`reduce_by_index`** were used. This lead the program to crash because it creates a huge multi-dimensional array that cannot be fused. Furthermore, the current version discovered a bug in our implementation, because huge indices caused integer overflow.

Histo

The Histo benchmark program is a two-dimensional histogram of a two-dimensional input with higher density around the center of the histogram. In Parboil the implementation produces a BMP file which is not included in the timing, thus our implementation does not create a BMP file in the first place.

The corresponding Futhark implementation showcases the simplicity of the function exposed to the user:

```
let main (img_width : i32) (img_height : i32)
         (his_width : i32) (his_height : i32)
         (img : []i32) : []i32 =
  reduce_by_index (replicate (his_width * his_height) 0)
                  (+) 0 img
                  (replicate (img_width * img_height) 1)
```

compared to the Parboil OpenCL implementation, which you need to be rather proefficent in OpenCL to write.

Figure 37 displays the results, where **REF** is the OpenCL NVIDIA implementation from Parboil, displaying runtime in microseconds, and **FUT** is the Futhark implementation displaying speedup. We see that our implementation on

this particular dataset is very efficient, and we conjecture that there are at least two reasons for this:

- Even though the description says that the input data has a higher density around the center of the histogram, we see from the dataset that the resulting histogram has dimensions 256×4096 which is over one million buckets in a one-dimensional representation.
- The Parboil OpenCL implementation divides the histogram into a grid, such that groups of thread blocks cooperate on a subhistogram and each thread block in the group is responsible for a range of buckets. This strategy requires the input to be sorted which has a cost. It also appears that each element may be processed multiple times because each thread handles only a certain range of buckets in the subhistogram.

In total, we conjecture that the overhead in the baseline implementation caused by manipulations are simply not amortized on this dataset, while it may be efficient on datasets with higher density. On the contrary, the Futhark program is computing a large histogram, which we think looks like multiple **D4** datasets from the previous section in succession, i.e., it has multiple high-density regions due to the flattening.

6.2.2 Rodinia

The Rodinia benchmark suite contains the classical k-means clustering algorithm, which is about grouping points in a d-dimensional space into k clusters, such that each cluster contains the points that are closest to its center. For each cluster the centre is the mean of all points in the cluster.

In order to compute the mean, we must know the number of points in a cluster, which can be seen as a histogram. The current Futhark implementation uses the **`stream_red_per`** construct for this which is shown in Figure 39(a). The sequential histograms are computed in lines 4-6 and produces partial results of size k, which is the number of clusters (or buckets if you will). These partial results are then summed using the **`map2`**-function with the addition operator. The rewritten program is shown in Figure 39(b).

Figure 38 displays the results, where **REF** is the Rodinia OpenCL implementation which acts as baseline and displays its runtime in milliseconds. The row **FV1** refers to the Futhark implementation using **`stream_red_per`**, and **FV2** refers to the rewritten program using **`reduce_by_index`**. Both display speedups with respect to **REF**. For both datasets, we see that the rewritten program is only slightly faster than the current. This is due to the number of clusters, k, being very small, such that the overhead from creating a histogram per thread

	D1	**D2**
REF (ms)	1255ms	1885ms
FV1 (×)	4.9×	5.1×
FV2 (×)	5.4×	7.2×

Figure 38: k-means benchmark. Speedup of two Futhark programs (**FV1** and **FV2**) compared to an OpenCL baseline implementation (**REF**) from the Rodinia benchmark suite, compared on two datasets (**D1** and **D2**) also from the Rodinia benchmark suite. Row **REF** is the baseline and displays runtime in miliseconds; **FV1** and **FV2** display speedups.

```
1  let points_in_clusters =
2    stream_red_per
3      (\(acc: [k]i32) (x: [k]i32) -> map2 (+) acc x)
4      (\(inp: []i32) ->
5           loop acc = (replicate k 0) for c in inp do
6           unsafe let acc[c] = acc[c] + 1 in acc)
7      membership
```

(a) Original part of the k-means program, responsible for counting the number of points belonging to each cluster.

```
let points_in_clusters =
  reduce_by_index (replicate k 0)
                  (+) 0 membership
                  (replicate n 1)
```

(b) Rewriting the part of the k-means program that counts the number of points in each cluster.

Figure 39: Rewriting part of the k-means program.

in **FV1** is insignificant. In the datasets k is maximally 8. If we instead had a dataset with a larger k we would see that **FV2** would significantly outperform **FV1**.

Part III

Final Remarks

7 Conclusion and Future Work

We have presented a new language construct for efficient computation of generalised reductions in the programming language Futhark. Its implementation is based on atomic functions and the idea of subhistogramming, effectively letting GPU threads cooperate on partial results. To mitigate the effect of serialization caused by atomic functions we have developed a simple heuristic for determining the number of cooperating threads.

In practice, generalised reductions often manifest themselves as traditional histograms, and the implementation has been optimized to this specific case. In fact, the compiler chooses between three code generations, choosing the most efficient based on a combining operator provided by the user.

We have demonstrated that the runtime performance of the new construct is at least as good, and often much better, than existing solutions for computing traditional histograms in Futhark. The construct was tested on a collection of 12 adversarial datasets and we see speedups ranging from $\times 1.62$ to $\times 17.63$ compared to a sequential baseline, where existing solutions in Futhark show only moderate speedups and often slowdowns. In addition, we have rewritten two existing Futhark benchmarks to use the new construct, showing a competitive runtime performance.

7.1 Limitations and Future Work

The two rewritten Futhark benchmarks, TPACF and k-means, suggests that if the number of buckets in a histogram is small (< 20) existing solutions perform slightly better. Thus, although our new construct aims at being flexible, there are cases where its performance is known to be poor, e.g., very large histograms where only a few buckets are hit, and in such cases other methods may be used instead.

An interesting topic would be to investigate the cache behavior with respect to random writes, along with mitigation strategies. For example, we see a slowdown if indices are normally distributed around a mean bucket, compared to the case where all indices are hitting the same bucket.

One straightforward method for mitigating the impact on performance of random writes, is to move subhistogramming to shared memory instead of global memory. But since shared memory is limited we would need to generate two kernels at compile-time; one using shared memory and another using global memory. The global memory version can then be used as a fallback kernel whenever the histogram is too large to fit in shared memory. This optimization is also appealing as it is cheap in terms of introducing complexity in the compiler; a shared memory implementation would roughly follow the global memory implementation, and then we can generate both kernels by inserting an if-statement to choose a runtime. This is what is currently happening for regular segmented reductions [LH17].

Bibliography

[acc] Accelerate - High-Performance Parallel Arrays for Haskell. http://www.acceleratehs.org/.

[BS12] Shawn Brown and Jack Snoeyink. Modestly faster histogram computations on gpus. In *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*, pages 1–7, 2012.

[EH18] Martin Elsman and Troels Henriksen. *Parallel Programming in Futhark*. 2018.

[Eil14] Marco Eilers. Multireduce and Multiscan on Modern GPUs. Master's thesis, University of Copenhagen, Department of Computer Science, 2014.

[Hen] Troels Henriksen. Futhark Programming Language Homepage. https://futhark-lang.org/. [Online; accessed 05-October-2018].

[Hen17] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Department of Computer Science, 2017.

[HLO16] 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*, ARRAY 2016, pages 17–24, New York, NY, USA, 2016. ACM.

[Hov18] Anders Kiel Hovgaard. Higher-order functions for a high-performance programming language for gpus. Master's thesis, University of Copenhagen, Department of Computer Science, 2018.

[HSE+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[Jø16] Asbjørn Viderø Jøkladal. Implementing Discriminator and Generalized Histogram on GPGPUs. Master's thesis, University of Copenhagen, Department of Computer Science, 2016.

[LH17] Rasmus Wriedt Larsen and Troels Henriksen. Strategies for regular segmented reductions on gpu. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 42–52, New York, NY, USA, 2017. ACM.

[McD15] Trevor L. McDonell. *Optimising Purely Functional GPU Programs*. PhD thesis, University of New South Wales, School of Computer Science and Engineering, 2015.

[NvdBCM11] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Bart Mesman. High performance predictable histogramming on gpus: Exploring and evaluating algorithm trade-offs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[Pod07] Victor Podlozhnyuk. Histogram calculations in cuda. NVIDIA Cooperation: `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf`, 2007. [Online; accessed 17-October-2018].

[SK07] Ramtin Shams and R. A. Kennedy. Efficient histogram algorithms for nvidia cuda compatible devices. In *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, 2007.

[thr] Thrust. https://thrust.github.io/.

Appendices

A Prototyping Experiment

A.1 Experiment – Raw Data

The following runtimes are reported in milliseconds for space issues. Note, that a dash instead of a number indicates that the device had insufficient memory for generating the amount of subhistograms required by the given cooperation level.

Buckets: 16		**Cooperation level**								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	2.90	0.56	0.39	0.35	0.34	0.44	0.95	2.91	2.98
	Shared	0.90	0.43	0.45	0.55	0.75	1.54	-	-	-
CAS	**Global**	3.31	1.27	1.06	1.44	4.21	21.64	126.08	833.50	4265.64
	Shared	1.47	0.97	1.50	2.88	7.73	44.21	-	-	-
Xchg	**Global**	8.84	2.05	1.77	3.06	7.59	23.52	156.82	2158.17	4728.41
	Shared	4.32	2.88	2.53	4.56	10.22	38.87	-	-	-
Futhark		0.11	0.05	0.04	0.04	0.03	0.02	0.02	0.02	0.02

Buckets: 64		**Cooperation level**								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	7.84	2.59	0.56	0.45	0.46	0.43	0.53	1.29	2.39
	Shared	2.91	0.93	0.44	0.43	0.46	0.70	-	-	-
CAS	**Global**	12.47	2.82	1.32	1.18	1.84	5.95	18.89	125.31	860.12
	Shared	5.08	1.89	1.02	1.34	2.13	5.57	-	-	-
Xchg	**Global**	25.09	8.60	2.13	2.06	3.62	8.48	25.08	254.87	1058.88
	Shared	16.71	5.88	3.05	2.34	3.88	8.89	-	-	-
Futhark		0.29	0.10	0.05	0.04	0.04	0.02	0.02	0.02	0.02

Buckets: 256		Cooperation level								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	9.14	6.60	2.44	0.68	0.69	0.67	0.61	0.71	1.08
	Shared	13.15	2.94	0.93	0.45	0.45	0.54	-	-	-
CAS	**Global**	13.70	10.16	2.75	1.82	2.18	3.57	7.18	20.44	101.71
	Shared	21.49	5.55	1.96	0.98	1.20	1.96	-	-	-
Xchg	**Global**	26.67	20.87	8.42	2.91	3.89	6.24	12.75	35.28	127.85
	Shared	68.20	18.60	6.15	2.96	2.31	3.88	-	-	-
Futhark		1.28	0.54	0.33	0.05	0.05	0.02	0.02	0.02	0.02

Buckets: 1024		Cooperation level								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	10.16	8.71	6.37	2.79	0.86	0.88	0.82	0.79	0.80
	Shared	82.27	13.20	2.95	0.94	0.47	0.46	-	-	-
CAS	**Global**	15.10	13.08	10.27	3.51	3.34	3.83	5.30	10.35	17.09
	Shared	114.11	21.91	5.65	1.83	0.89	1.12	-	-	-
Xchg	**Global**	28.88	25.48	20.17	11.77	5.70	6.79	9.31	16.60	27.89
	Shared	292.15	70.12	18.89	5.87	2.90	2.30	-	-	-
Futhark		4.55	1.12	0.68	0.13	0.06	0.04	0.02	0.02	0.02

Buckets: 4096		Cooperation level								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	12.70	10.05	8.67	6.91	3.17	0.94	0.91	0.89	0.90
	Shared	751.64	82.27	13.20	2.96	0.94	0.47	-	-	-
CAS	**Global**	17.59	14.88	13.27	11.08	4.36	4.21	4.61	5.98	7.59
	Shared	856.90	114.22	22.02	5.50	1.73	0.87	-	-	-
Xchg	**Global**	34.04	28.58	25.46	22.23	15.82	7.41	8.50	10.55	13.39
	Shared	2851.49	413.28	100.75	27.79	8.36	2.91	-	-	-
Futhark		19.19	4.69	1.19	0.37	0.18	0.08	0.03	0.02	0.02

Buckets: 16384		Cooperation level								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	-	12.68	10.01	8.86	7.34	3.29	0.95	0.94	0.96
	Shared	-	-	-	-	-	-	-	-	-
CAS	**Global**	-	17.59	14.82	13.45	11.49	4.62	4.47	4.83	5.23
	Shared	-	-	-	-	-	-	-	-	-
Xchg	**Global**	-	34.00	28.38	25.91	23.52	16.81	8.19	8.89	9.72
	Shared	-	-	-	-	-	-	-	-	-
Futhark		-	19.70	4.81	1.49	0.66	0.28	0.06	0.03	0.02

Buckets: 61440		Cooperation level								
		1	**4**	**16**	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**Global**	-	-	12.49	9.98	8.99	7.38	3.18	1.02	0.96
	Shared	-	-	-	-	-	-	-	-	-
CAS	**Global**	-	-	17.45	14.88	13.77	11.56	4.79	4.54	4.66
	Shared	-	-	-	-	-	-	-	-	-
Xchg	**Global**	-	-	33.63	28.57	26.80	24.20	16.88	8.27	8.67
	Shared	-	-	-	-	-	-	-	-	-
Futhark		-	-	18.46	5.28	2.41	1.02	0.20	0.06	0.03

A.2 Experiment – Graphs

The following graphs are based on the raw data presented in the previous appendix. Note that the runtimes here are report in microseconds for visualization purposes.

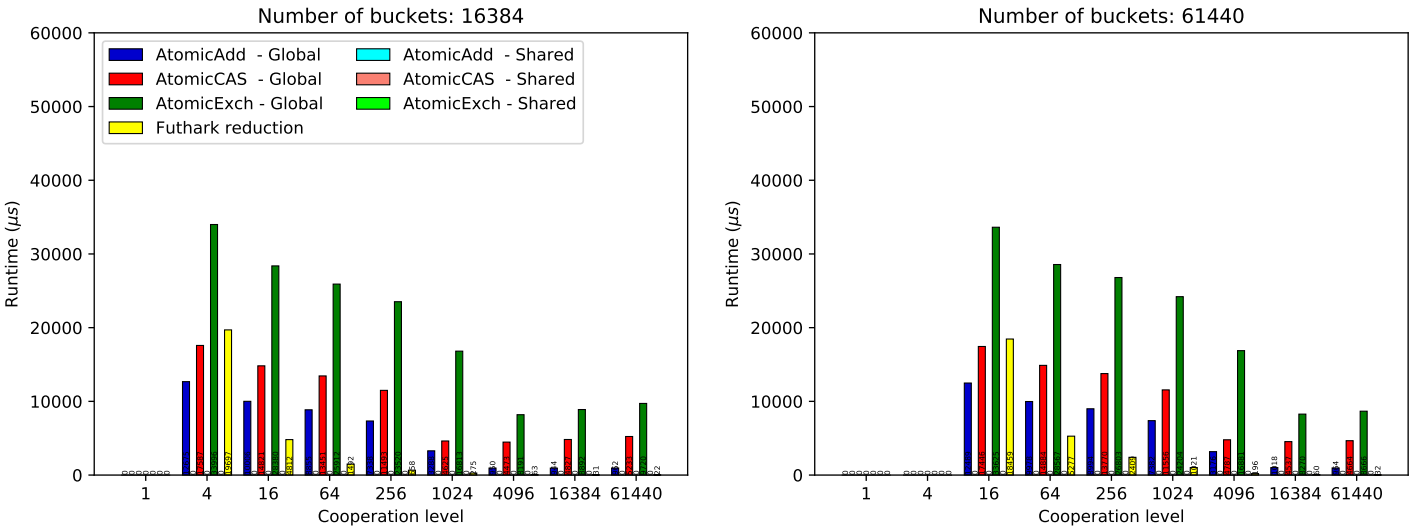


Figure 40: Runtimes in microseconds for the three locking strategies in global memory, including the reduction phase in Futhark. These histogram sizes does not fit in shared memory.

A.3 Experiment – Subhistogramming Data

See Table 3.

			16	**64**	**256**	**1024**	**4096**	**16384**	**61440**
Add	**REF**	(coop. level)	256	1024	4096	16384	16384	16384	61440
		(μs)	372μs	453μs	629μs	808μs	913μs	976μs	997μs
	Global	($\times$)	0.86$\times$	0.94$\times$	0.85$\times$	0.89$\times$	0.96$\times$	1.00$\times$	1.00$\times$
	REF	(coop. level)	4	64	64	1024	1024	-	-
		(μs)	483μs	467μs	500μs	500μs	555μs	-	-
	Shared	($\times$)	0.99$\times$	1.00$\times$	0.99$\times$	1.00$\times$	1.00$\times$	-	-
CAS	**REF**	(coop. level)	16	64	64	256	1024	4096	16384
		(μs)	1103μs	1217μs	1871μs	3405μs	4296μs	4537μs	4598μs
	Global	($\times$)	1.00$\times$	1.00$\times$	0.84$\times$	0.88$\times$	0.92$\times$	0.93$\times$	0.98$\times$
	REF	(coop. level)	4	16	64	256	1024	-	-
		(μs)	1022μs	1073μs	1025μs	956μs	952μs	-	-
	Shared	($\times$)	0.66$\times$	0.78$\times$	0.82$\times$	0.82$\times$	1.00$\times$	-	-
Xchg	**REF**	(coop. level)	16	64	64	256	1024	4096	16384
		(μs)	1812μs	2094μs	2960μs	5766μs	7495μs	8255μs	8331μs
	Global	($\times$)	1.00$\times$	1.00$\times$	0.75$\times$	0.84$\times$	0.88$\times$	0.93$\times$	0.96$\times$
	REF	(coop. level)	16	64	256	1024	1024	-	-
		(μs)	2570μs	2383μs	2363μs	2333μs	2993μs	-	-
	Shared	($\times$)	1.00$\times$	1.00$\times$	1.00$\times$	1.00$\times$	1.00$\times$	-	-

Table 3: Subhistogrammin data.

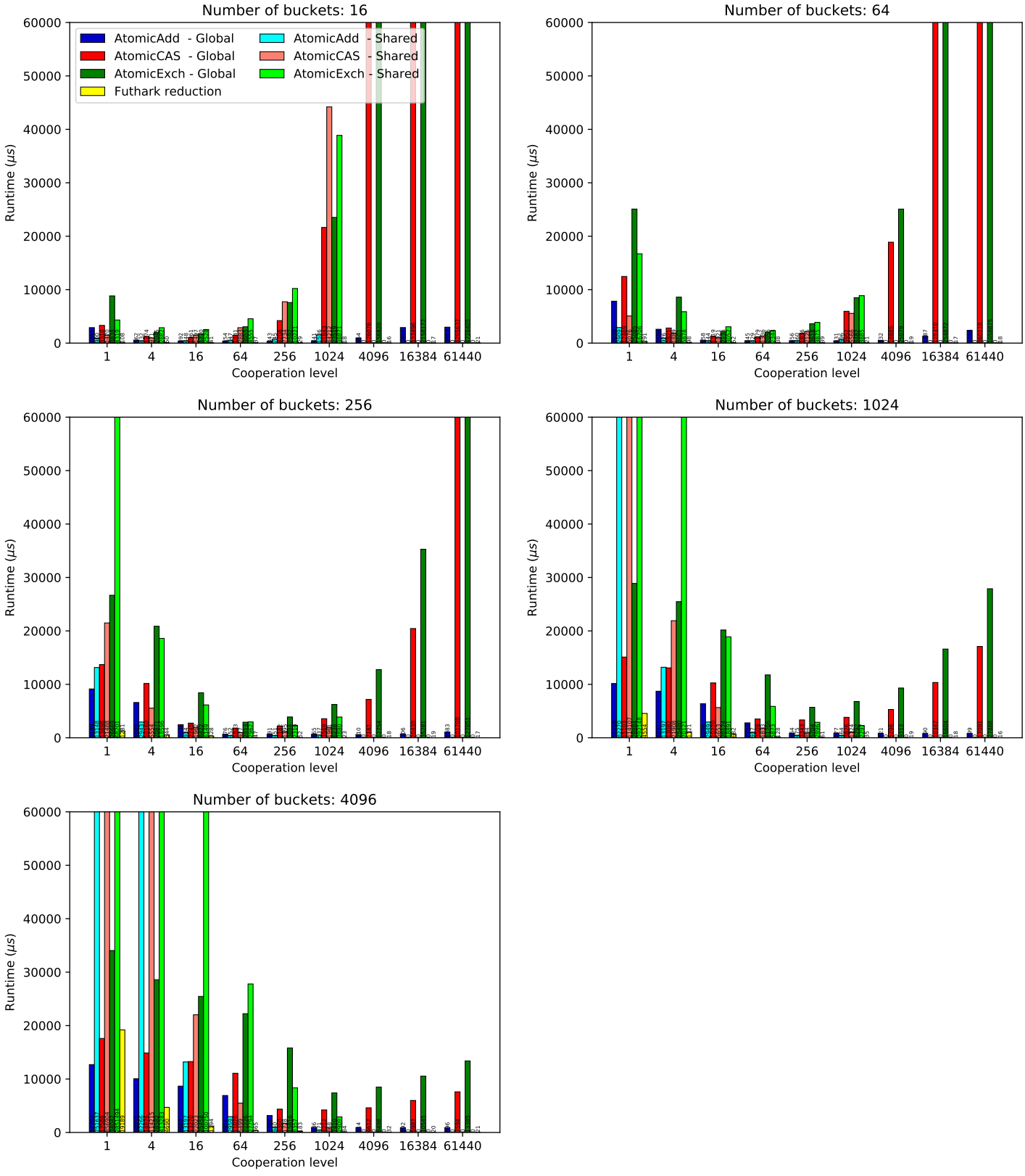


Figure 41: Runtimes in microseconds for the three locking strategies in both global and shared memory, including the reduction phase in Futhark..

B Implementation

B.1 Visualization of Subpasses in Middle Stage

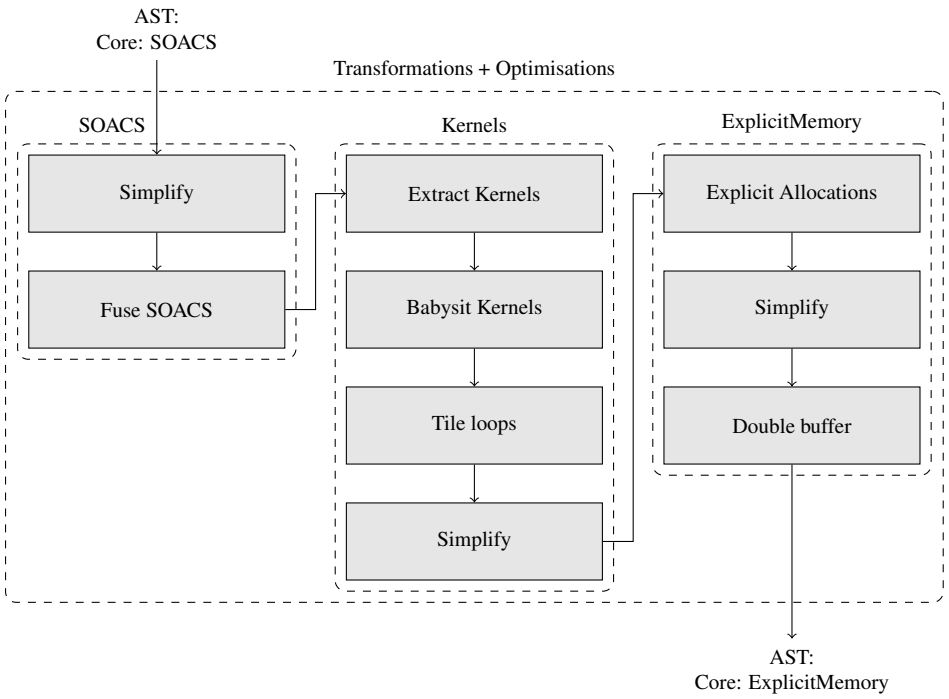


Figure 42: Overview of transformations and optimisations. Note, that this is only a selection of the actual passes performed by the compiler, and that there may be passes in between the displayed passes. In particular, the program is type checked after each change of representation.

B.2 Call-graphs for Back End

To assist our understanding of the back end we created two call-graphs; one for `futhark-opencl` and one for `futhark-c`, which are displayed in Figures 43 and 44, respectively. Both graphs contain only selected calls pertaining to the back end and we have omitted all calls to functions in the middle and front end.

When mapping the boxes from Figure 43 to the high-level diagram from the beginning of the back end section, we have that box eight corresponds to box a, box seven corresponds to box b, and box five corresponds to box c.

Comparing the two call-graphs we see that the one for `futhark-opencl` contains two additional boxes; boxes six and seven. Roughly, we can say that they are responsible for adding the notation of calls to kernels in the intermediate code (box six); for augmenting an intermediate program with a string containing the OpenCL programs, and for adding calls to these kernels (box seven).

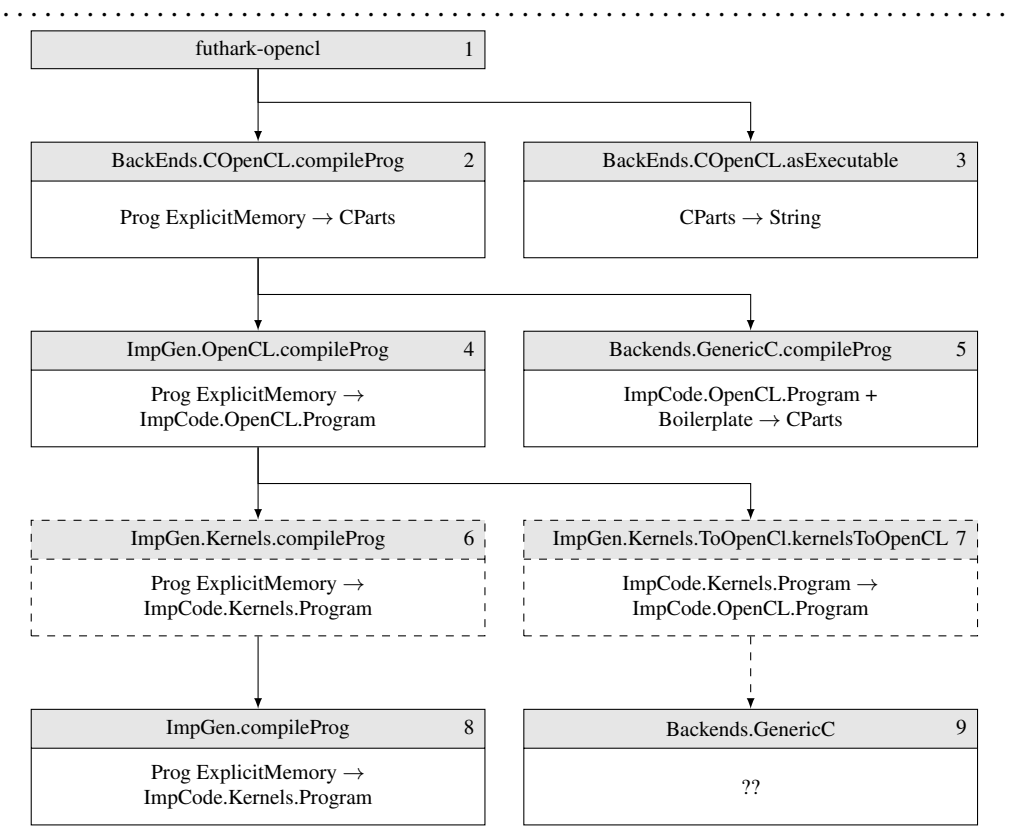


Figure 43: Call-graph for `futhark-opencl`. (Selected calls.) Shaded parts refer to Haskell functions, while non-shaded parts show input and output types for the function. Dashed boxes are the difference compared to the call-graph for `futhark-c` shown in Figure 44.

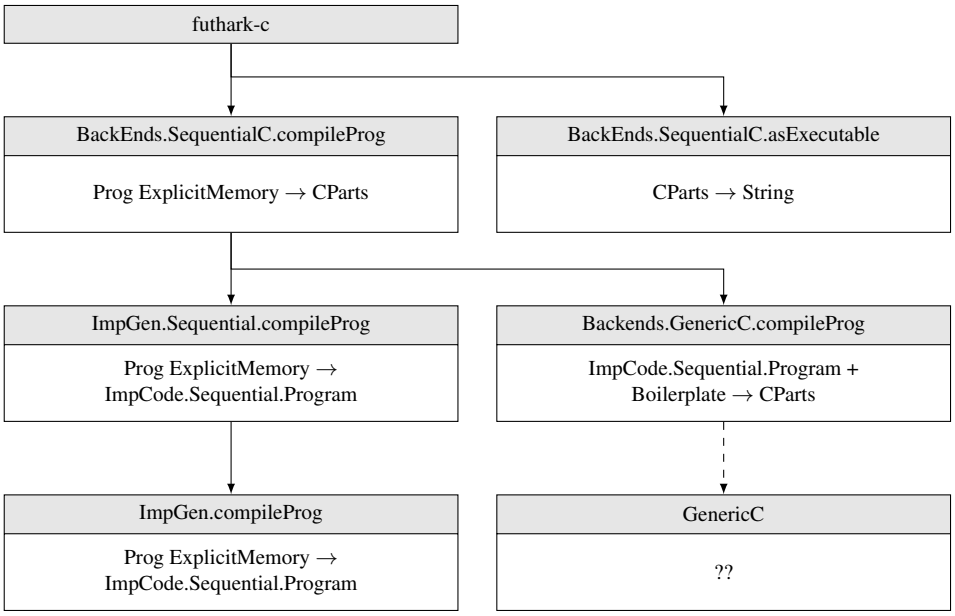


Figure 44: Call-graph for `futhark-c`. (Selected calls.) Shaded parts refer to Haskell functions, while non-shaded parts show input and output types for the function.