Master's Thesis

Multi-GPU Futhark Using Parallel Streams

Steffen Holst Larsen

Supervisors: Cosmin Eugen Oancea & Troels Henriksen

September 19, 2019

Abstract

In this thesis we describe the design and implementation of a new internal streaming operator in the Futhark compiler called the *husk operator*. The intention of the husk operator is to express the distribution of an operation, ultimately allowing compiled programs to utilize multiple graphics processing units (GPUs) in the system they are executed on. Since the husk operator is internal to the compiler, the usage of it is automatically generated through transformations of select existing second-order array combinators in the Futhark programming language.

We start out by introducing the type and semantics of the husk operator, which are then used to introduce transformations for the map and reduce SOACs of Futhark, relating their semantics to that of the husk operator.

We then discuss, at a high level, the stages of the compiler that is adjusted to accommodate the husk operator, introducing the transformation between the intermediate representations of the husk operator. For the backend of the compiler, we focus on changes made to the CUDA C backend, introducing a worker-thread runtime environment for isolated execution of the individual execution of the operations contained in the husk operators. Since the husk operator is agnostic to the number of GPUs being used, it is also present when only a single GPU is to be used, so following the changes to the CUDA C backend of Futhark we discuss the aspects of the husk operator implementation that may affect the performance of Futhark programs using a single GPU and how these are mitigated.

Finally we evaluate the performance of the husk operator in two parts. First we focus on Futhark programs that are executed using only a single GPU, comparing the performance when using the husk operator to the performance of the same program not using the husk operator, analyzing examples on both ends of the spectrum of relative performance. Secondly we compare select established benchmark programs when running with one and two GPUs, analyzing cases in which the husk operator introduces good scaling with large data set, as well as cases where the inter-GPU data transfer overhead becomes too expensive relative to the computations, giving undesirable performance.

Contents

Appendices

1 Introduction

In modern computer systems the graphics processing unit (GPU) has expanded its capabilities outside the processing of graphics tasks and has become an invaluable tool for data-heavy computations. With its many cores and dedicated architecture, the GPU can exploit the parallelism of a problem to gain large performance increases relative to the same problem on a CPU, depending on the problem at hand. In order to do the computations the GPU acts as a co-processor to the CPU, which means that the CPU is in control of the computations to be done on the GPU and therefore has to make sure that the relevant data is present on the GPU before launching kernels on the GPU, the result of which may then be returned to the CPU to be used by the program for further execution.

Though modern computer systems generally have a single GPU to utilize, some systems need more resources than even a single state-of-the-art GPU may offer and may therefore incorporate multiple. By having multiple GPUs, the CPU gains additional co-processors, which in turn adds more opportunity for higher performance by exploitation of the additional resources. However, more GPUs also complicates the execution of a program as it must be distributed both in parallel tasks and in memory, to account for the individual memory spaces introduced by each of the GPUs. Since a problem may also require intercommunication and synchronization between the individual parallel tasks of the corresponding program, the difficulty and added overhead of such operations between GPUs may simply be unfeasible, thus not all problems that are efficient on a single GPU are suitable for multiple GPUs.

In order to allow programmers to write programs that can utilize the compute capabilities of the GPUs in a host system, a number of compute APIs allows the interfacing between the CPU and the GPUs through API calls done in high-level programming languages such as C and Python. Examples of such compute APIs are the OpenCL API and the CUDA API. However, not only is the additional groundwork and book keeping needed to do the interfacing tedious, it often also requires specialized knowledge about compiler analysis and the architecture of the GPUs in use. To alleviate the programmers, the Futhark programming language enables the writing of high-level, hardware-agnostic, data-parallel programs which are compiled to host programs that efficiently handles the interfacing between the CPU and GPU, greatly improving productivity and maintainability, all the while allowing the programmer to stay oblivious to the intricacies of the underlying hardware. However, leaving the compiler to create the use of the compute APIs also leaves the programmer at the mercy of the abilities and limitations of the compiler they use. For the Futhark language, one such limitation is that the compiled programs can only use a single GPU on the host system.

Some programs in Futhark may however be suitable for distribution amongst multiple devices. Assuming arrays `inds:[M]i32`, `xss, zss:[M][N]f32` and `ys:[N]f32`, consider the Futhark expression in Listing 1 using the `map` array combinator. Since the individual applications of the lambda functions to the

Listing 1 A Futhark expression suitable for distribution between devices.

```
1  map (\xs ind ->
2       let s1 = reduce (+) 0 zss[ind]
3       let s2 = map (*) xs ys |> reduce (+) 0
4       in  (s1 + s2)
5     ) xss inds
```

outer level of array `xss` are independent of each other, it could be distributed between multiple GPUs. However, since GPUs have their own memory spaces, the data of the arrays must also be transferred to all devices. A naive solution would be to broadcast all arrays used by the operation to all devices, but since such memory operations can be very expensive, minimizing the intercommunication could prove crucial to actually have multiple GPUs increase the performance of a Futhark program. For the Futhark expression in Listing 1, the data would be optimally distributed as:

(a) The entire array `ys` is accessed by each GPU, hence it must be broadcast.

(b) Each GPU would access non-overlapping slices/rows of `xss`, hence the compiler can optimize communication by copying to each GPU precisely those slices; this pattern is simple to exploit given that the `xss` array is directly mapped, hence we do not have to look at the body of the lambda function.

(c) Each GPU accesses various slices/rows of `zss`, but based on an indirect array `inds`; this can be optimized in the same way as `xss`, but more analysis is necessary: Since `zss` in not directly mapped, the compiler would need to analyze the body of the lambda function to determine the slice of `zss` which is accessed by each GPU. This information can be extracted by an inspector, i.e. a piece of code that is executed before the `map` operation and computes the to-be-communicated chunks at runtime.

The focus and main contribution of this thesis is a streaming operator in the Futhark compiler which allows for the distribution of a program and the corresponding data amongst multiple GPUs. To keep focus on the automated transformations of existing operators inside the Futhark compiler to semantically equivalent streams to be distributed between devices, this streaming operator is only made internal to the compiler, thus not allowing explicit use in the Futhark language for now, though it could be added to the language in the future. Throughout the stages of the compiler, this streaming operator is transformed and optimized before finally reaching the backend. With respect to minimizing transference of data, we will focus on cases (a) and (b) as they can be determined at compile-time, leaving case (c) for future work.

Following the introduction of the changes to the compiler stages with the addition of the intermediate representations of the streaming operator, we discuss

the changes to the CUDA C backend of the Futhark compiler needed in order to be able to use multiple GPUs. We then introduce a worker-thread runtime environment, which utilizes the multi-thread capabilities of modern CPUs to minimize the host-based overhead of the additional operations required when working with multiple GPUs. The worker-thread runtime environment can be seen as a secondary contribution of the thesis, as it can be generalized for other multi-core purposes in the future.

We then compare the single-GPU performance with the streaming operator to the single-GPU performance of the same programs compiled using the original Futhark compiler, in order to make sure that the ability to use multiple GPUs does not come at the cost of worse performance on the much more common single GPU case. We then compare the single-GPU performance with the multi-GPU performance of select programs, illustrating both Futhark programs that are suitable and unsuitable for execution on multiple GPUs, analyzing the causes of the suitability using profiling tools.

1.1 Thesis Overview

To give a high-level overview of the primary aspects of this thesis, we summarize the most important points:

- We focus on the distribution of map and reduce array combinators across multiple GPUs by applying transformations to the corresponding operations in Futhark programs, creating a new internal streaming operator that express the distribution of the operation and their input data.
- We use a centralized execution model with a master GPU, also referred to as GPU 0, that must contain all device-side data outside the execution of the new streaming operator. Upon execution of the streaming operator, all relevant data is transferred to the other GPUs, whereas the results are either concatenated on the master GPU or combined on the host using a reduce operation.
- We implement the streaming operator for the backend of the Futhark compiler that generates C code utilizing the CUDA API and attempt to hide the overhead of working with multiple GPUs by use of a multi-threaded runtime environment using message passing to communicate.
- We discuss and implement a number of optimizations for the new streaming operator for cases where only a single GPU is used, reducing the introduced overhead.

Part I
Bird's-Eye View

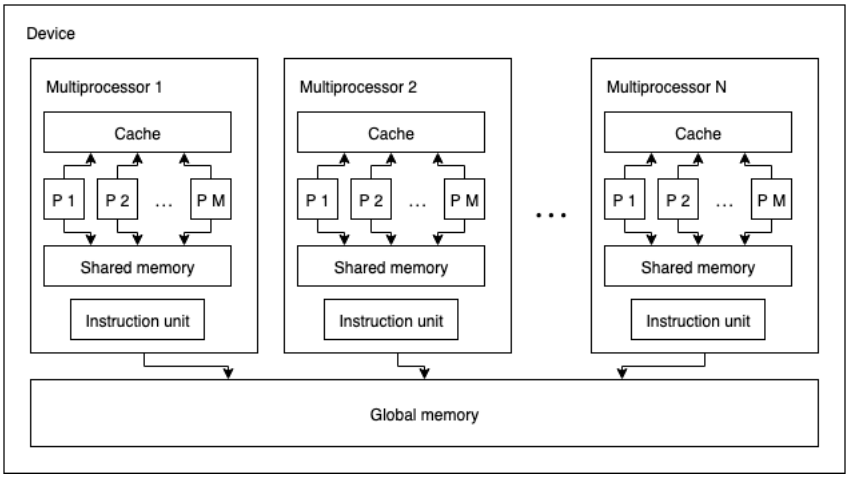

Figure 1: General GPGPU architecture. Each square marked with P represents a processor of the encapsulating multiprocessor.

2 Background

The focus of this section is to give the reader the necessary background information required to understand the details in the rest of the paper. We start out by exploring, at a high level, the GPGPU architecture and discuss the effects of having multiple GPUs working in tandem on the same system. We then introduce the Futhark programming language, focusing on the most relevant features with respect to the thesis itself.

2.1 GPGPU Architecture

Though the exact architecture of the compute section of modern GPUs can vary greatly between different generations and manufacturers, a general architecture of GPGPUs can be summarized in Figure 1 [28]. Inside the device we find N multiprocessors, which in turn contains M processors. Note that, though it is not illustrated, each processor has its own registers. The architecture also infers a memory hierarchy consisting of 3 levels:

1. Registers are exclusive to their corresponding processor. This is the fastest memory space.

2. Shared memory is exclusive to each multiprocessor, but is shared between its processors. Likewise, the multiprocessors have a varying number of cache levels, some of which are also shared between the processors.

3. Global memory is accessible by all processors on all multiprocessors. This is the slowest memory space, but also typically largest memory space. In

some cases there may also be global caches shared between all processors on all multiprocessors.

Built upon this, the multiprocessors each have an instruction unit, allowing a sub-grouping of their processors to execute in lockstep through the use of SIMD instructions. For example, these sub-groups are known as warps on modern NVIDIA GPUs, which typically consists of 32 processors.

To gain a better understanding of modern GPUs we draw concrete example from the NVIDIA GeForce RTX 2080, which is built upon the state of the art NVIDIA Turing architecture. With respect to the GPGPU architecture, this discrete GPU, much like other modern NVIDIA GPUs, follows the CUDA model. The RTX 2080 model has a total of 46 streaming multiprocessors, each with 64 cores, and boasts 8 GB of global memory. Each streaming multiprocessor has a 256 KB register file, from which threads allocate their registers from, as well as 96 KB of combined L1 cache and shared memory, allowing the cache to grow when the full capacity of the shared memory is not being utilized.

2.1.1 Multi-GPU Systems

In most modern general-purpose computer systems the GPU comes in many shapes, be it as part of an APU such as the AMD Ryzen 5 2400G or be it a discrete GPU such as the aforementioned NVIDIA GeForce RTX 2080, but most commonly they act as a co-processor to the CPU. From here we will at times refer to the CPU and the systems memory as the *host*, whereas the GPUs of a system will be referred to as the *devices*.

To execute a program on a GPU the host is typically required to transfer a program to the GPU, often referred to as a *kernel*. Additionally the host must transfer any data required for the computation ahead of launching the kernel.

Likewise, some general-purpose computer systems may incorporate multiple GPUs, which mostly require the same procedure in executing a kernel as with a single GPU. However, compared to only using a single GPU, the challenge when introducing n GPUs into the GPU computational model is distributing the problem and the data between the devices. In the optimal cases this can give a factor n speedup and potentially a factor n reduction of memory usage on each device. For some problems this may introduce expensive memory transfers or even inter-device synchronizations, which cannot be distributed without extremely expensive host-device synchronization, rendering such approaches infeasible.

Since GPUs in multi-GPU systems have individual memory spaces, there can be cases where some of the data residing on a GPU G_1 is needed on a GPU G_2, requiring an inter-GPU memory copy operation to transfer the data from G_1 to G_2. Such situations could for example arrive when a kernel to be executed on G_2 needs a part of the result of a previously executed kernel that resides on G_1. The simple approach for accomplishing inter-GPU memory copies is to first copy the data from G_1 to the host memory and then to G_2, but due to the relatively low bandwidth of the interconnect between the GPUs and the CPU,

this is often a bottleneck for such operations. To alleviate this, other high-speed interconnects have been developed specifically to allow direct transfer between GPUs in a system.

As an example of a direct high-speed interconnect, we consider the NVIDIA NVLink bridge for NVIDIA GeForce RTX 2080 and 2080ti GPUs, which is a physical extension bridge for systems that incorporate two of these GPUs. To understand the importance of this bridge, consider two NVIDIA GeForce RTX 2080 GPUs, the specifications of which were mentioned previously. These GPUs support up to 16 lanes on a PCIe 3.0 bus which in turn offers about 1 GB/s bandwidth per lane, whereas the NVLink instead offers a bandwidth of 25 GB/s in each direction. This means that using an NVLink bridge between two GPUs offers $\sim$ 56% higher bandwidth while preventing congestion on the PCIe bus. This assumes that each GPU uses 16 PCIe lanes, which may not be the case due to limitation on the number of PCIe lanes of the host CPU, so the bandwidth difference may be even higher on some systems. Notice however that these bandwidths are still magnitudes slower than the internal memory bandwidth of the GPU, e.g. 448 GB/s on the NVIDIA GeForce RTX 2080.

2.2 Futhark Programming Language

Futhark is a monomorphic, statically typed, strictly evaluated, purely functional data-parallel programming language exposing a selection of parallel bulk array operators. Though Futhark can target similar platforms as more established programming languages, such as Haskell or C, its strength lies in its ability to allow for the utilization of massively parallel hardware, such as modern GPUs, through the use of compute APIs such as CUDA and OpenCL, all the while allowing the programmer to be oblivious to the intricacies of the underlying hardware [17] and supporting integration with mainstream productivity-oriented environment such as Python [15].

In this section we will give the reader a very basic understanding of the Futhark programming language, giving a brief introduction to the syntax of a Futhark program, specifying some of the relevant features of the language, and introducing the operators that are the most important for this thesis. This section is based on [17, 9, 14].

2.2.1 Futhark Programs

Syntactically, Futhark is similar to popular functional programming languages, such as SML and Haskell. A simple example of a Futhark program for computing the dot product of two vectors `x` and `y` , i.e. $\sum_i \mathtt{x}_i \cdot \mathtt{y}_i$, is shown in Listing 2. In this example a function `main` is defined with a number of parameters, representing the two vectors by the corresponding names. Notice that these vectors use the size parameter `m`, denoted as `[m]` before the type, with the same specified similar to parameters outside type notations. Inside the main function we use two of the built-in operators of the Futhark language, namely `reduce` and `map2`, which are both examples of array combinators in Futhark.

Listing 2 A simple Futhark program.

```
1 let main [m] (x: [m]i32) (y: [m]i32): i32 =
2   let z = map2 (*) x y
3   in reduce (+) 0 z
```

Futhark accomplishes parallelism through a concept called explicit data-parallelism, that is the parallelism of the program is specified through the built-in operators, such as `reduce` and `map`. Consider again the program in Listing 2, where the operator `map2` on line 2 specifies that the multiplication operation between the elements of the vectors can be done in parallel, whereas the reduce operator specifies a reduction, which can also be done in parallel. The compiler chooses how to handle the parallelism given and may optimize and combine parallel operations as it sees fit. This exposes one of the primary strengths of the Futhark programming language, specifically that the user can use high-level operators for specifying the parallelism of a problem, whereas the compiler optimizes the parallelism and handles the low-level representation of the problem for the hardware at hand.

2.2.2 Array Operations

The key to good performance in Futhark is the use of the built-in array bulk transformation operators. These operators are categorized in two groups, namely first- and second-order array combinators. The first-order array combinators are operations on or resulting in arrays that always perform the same operation on the parameters. Examples of first-order array combinators are shown in Table 1 where $[m]\alpha$ for any integer m and Futhark type α denotes the array of m elements from α. Likewise for any Futhark types α and β, (α, β) denotes the composite type of 2-tuples. Notice that `zip` and `unzip` have variants that produce or takes lists with d-tuples for $d > 2$, where the operator name is suffixed with d, e.g. `unzip4` for 4-tuples.

In contrast to first-order array combinators, second-order array combinators (SOACs) takes a functional argument specifying the operation to be performed. The SOACs are the fundamental building blocks for writing programs in Futhark. Table 2 shows a selection of SOACs. As with `zip` and `unzip`, the `map` SOAC has variants suffixed by a number denoting the number of arguments of the functional argument, an example of which was shown through the use of `map2` in Listing 2.

Both `map` and `reduce` have special streaming variants, namely `stream_map` and `stream_reduce`, that applies the operations to sub-arrays of the input array, rather that the individual elements. The chunking may vary, but the size of the corresponding chunk is given as a parameter in the functional argument.

$\texttt{zip} : \forall m\, \alpha\, \beta\, .\, [m]\alpha \to [m]\beta \to [m](\alpha, \beta)$
$\texttt{zip}\, x\, y \equiv [(x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1})]$
The `zip` operator takes two arrays of equal size and produces an array of the same size containing pairs of the values of the input arrays in order.

$\texttt{unzip} : \forall m\, \alpha\, \beta\, .\, [m](\alpha, \beta) \to ([m]\alpha, [m]\beta)$
$\texttt{unzip}\, x \equiv ([x_0.1, x_1.1, \ldots, x_{m-1}.1], [x_0.2, x_1.2, \ldots, x_{m-1}.2])$
The `unzip` operator is the inverse of the zip operator, taking an array of pairs and separating them into two equally sized arrays containing the first and second elements of the pairs respectively.

$\texttt{iota} : (m : \texttt{int}) \to [m]\texttt{int}$
$\texttt{iota}\, m \equiv [0, 1, \ldots, m-1]$
The iota operator takes a single integer parameter m and produces an array of length m with values ranging from 0 to $m-1$.

Table 1: Types, semantics, and descriptions of a subset of the first-order array combinators in the Futhark programming language.

$\texttt{map} : \forall m\, \alpha\, \beta\, .\, (\alpha \to \beta) \to [m]\alpha \to [m]\beta$
$\texttt{map}\, f\, x \equiv [f\, x_0, f\, x_1, \ldots, f\, x_{m-1}]$
The `map` SOAC takes a univariate function and applies it to each element of the given array.

$\texttt{reduce} : \forall m\, \alpha\, .\, (\alpha \to \alpha \to \alpha) \to \alpha \to [m]\alpha \to \alpha$
$\texttt{reduce}\, \oplus\, \varepsilon\, x \equiv \varepsilon \oplus x_0 \oplus x_1 \oplus \cdots \oplus x_{m-1}$
The `reduce` SOAC combines, or reduces, the values of x using the monoid $(\oplus, \varepsilon)$.

$\texttt{scan} : \forall m\, \alpha\, .\, (\alpha \to \alpha \to \alpha) \to \alpha \to [m]\alpha \to [m]\alpha$
$\texttt{scan}\, \oplus\, \varepsilon\, x \equiv [\varepsilon \oplus x_0, \varepsilon \oplus x_0 \oplus x_1, \ldots, \varepsilon \oplus x_0 \oplus x_1 \oplus \cdots \oplus x_{m-1}]$
The `scan` SOAC computes the generalized prefix sum of the input array x with the monoid $(\oplus, \varepsilon)$. That is, each element at a given index i is the reduction of the slice $x[0 : i+1]$ using the monoid $(\oplus, \varepsilon)$.

Table 2: Types, semantics, and descriptions of a subset of the second-order array combinators in the Futhark programming language.

Listing 3 Futhark-like pseudo code definition of the semantics of the husk operator where m_i for $i \in [1, 2, \ldots, p]$ is the end index of a given partition of a.

```
1 husk (⊙) e f a ≡
2   let (xs, ys) =
3     unzip <| map f [m_1, m_2, ..., m_p]
4                    [m_1, m_2 − m_1, ..., m_p − m_{p−1}]
5                    [a[0:m_1], a[m_1:m_2], ..., a[m_{p−1}:m_p]])
6   in (reduce (⊙) e xs, flatten ys)
```

3 Husks - An Internal Streaming Operator

In this section we introduce the bread and butter of this thesis, namely the internal streaming operator we will refer to as a *husk*. Let $[\sigma]\phi$ denote the set of all arrays of size σ consisting of elements from the set ϕ. A husk is an operator with type

$$\texttt{husk} : \forall m\, \alpha\, \beta\, \gamma\, .\, (\alpha \to \alpha \to \alpha) \to \alpha \to ((m' : \texttt{i32}) \to \texttt{i32} \to [m']\beta \to (\alpha, [m']\gamma)) \\ \to [m]\beta \to (\alpha, [m]\gamma)$$

where `i32` denotes the 32-bit integer type in Futhark. Consider an input array $\texttt{a} \in [m]\beta$, a function $\texttt{f} : \texttt{i32} \to \texttt{i32} \to [m']\beta \to (\alpha, [m']\gamma)$, and a monoid $(\odot, \texttt{e})$, that is $\odot : \alpha \to \alpha \to \alpha$ is an associative operator and $\texttt{e} \in \alpha$ is the neutral element of $\odot$ over α. Using a Futhark-like pseudo code, the semantics of the husk operator can be expressed as in Listing 3 where m_i for $i \in [1, 2, \ldots, p]$ is the end index of a given partition of `a`, where $m_i < m_{i+1}$ and $m_p \equiv m$. These semantics illustrates that the husk operator essentially is a map-reduce composition, where a map with `f` is first applied to partitions of the input data, with one part of the results reduced using the monoid $(\odot, e)$ and the other concatenated. The first two parameters of the body lambda function of the husk are the element offset and the size of the array given respectively. Typically, the function `f` will either return a value in the first part of the tuple, leaving the second element of the result tuple as the empty list, or it will return a list in the second part of the tuple, leaving the first element of the result tuple as some unit value that will not be used. However, this generalization of the husk operator will make the necessary distinction between concatenated results and reduced results, since their types are different though they are similar. As a small but important detail we allow the Futhark-like pseudo language used for these semantics to use irregular arrays, as to allow partitions to be varying in size.

3.1 SOAC Transformations

Though the husk operator could in time be implemented as an explicit operator in the Futhark language, our focus is on introducing the husk operator as a

Listing 4 A map operation using an array $\texttt{a} \in [m]\beta$ and a function $\texttt{g} : \beta \to \alpha$.

```
1  map  g  a
```

Listing 5 Rewriting of the `map` operation in Listing 4.

$$
\begin{aligned}
&\texttt{map g a} \equiv [\texttt{g a}[0], \texttt{g a}[1], \ldots, \texttt{g a}[\texttt{m}-1]] \\
&\qquad \equiv [\texttt{g a}[0], \texttt{g a}[1], \ldots, \texttt{g a}[\texttt{m}_1 - 1]] \;\texttt{++} \\
&\qquad\quad [\texttt{g a}[\texttt{m}_1], \texttt{g a}[\texttt{m}_1 + 1], \ldots, \texttt{g a}[\texttt{m}_2 - 1]] \;\texttt{++} \\
&\qquad\quad \ldots \;\texttt{++} \\
&\qquad\quad [\texttt{g a}[\texttt{m}_{p-1}], \texttt{g a}[\texttt{m}_{p-1} + 1], \ldots, \texttt{g a}[\texttt{m}_p - 1]]
\end{aligned}
$$

construction only inside the Futhark compiler. Thus, to be able to actually utilize the husk operator they must be generated by the compiler, which is in turn done through the transformation of select existing SOACs, specifically `map` and `reduce`. We again use the Futhark-like pseudo language to rewrite the semantics of these SOACs, which stays unchanged as specified in Table 2.

Notice that both `map` and `reduce` SOACs have streamed variants, allowing for the corresponding operation over partitions of the input arrays, parameterized by the size of each chunk. These are not discussed here, but are handled very similar to their non-stream counterpart.

3.1.1 Map

Assume again an array $\texttt{a} \in [m]\beta$ and a function $\texttt{g} : \beta \to \alpha$, we now consider the use of `map` with `a` and `g`, as shown in the expression in Listing 4. Recalling the semantics of the `map` SOAC in Table 2 and using the independence between applications of `g`, the expression can be rewritten as shown in Listing 5 where m_i for $i \in [1, 2, \ldots, p]$ is the end index of a given partition of the input data, with $m_i < m_{i+1}$ and $m_p \equiv m$, and `++` denotes the binary concatenation operator. Notice that the concatenation of these partitions can be expressed using the `flatten` operator, giving way to further rewriting as shown in Listing 6 where we can easily realize that each list to be concatenated are in turn semantically equivalent to a `map` operation. This leaves us with a final rewriting shown in Listing 7 that in turn makes transformations based on the following observations, to bring the `map` operator equivalence to the same form as the semantics of the husk operator:

- A univariate function in a `map` operation can be rewritten to a multivariate function in a `map` on multiple arrays, where all parameters other than the original input are ignored. Consequently the transformation of a `map` operation does not explicitly use the offset or the size of the array inside a husk.
- Since the first value of the tuple is discarded, and assuming no side effects

Listing 6 Further rewriting of a `map` operation based on Listing 5.

```
1  map g a ≡ flatten [[g a[0], g a[1], ..., g a[m_1 − 1]],
2                     [g a[m_1], g a[m_1 + 1], ..., g a[m_2 − 1]],
3                     ...,
4                     [g a[m_{p−1}], g a[m_{p−1} + 1], ..., g a[m_p − 1]]]
```

Listing 7 Final rewriting of a `map` operator semantics, relating it to the husk operator.

```
1  map g a ≡ flatten [map g a[0 : m_1],
2                     map g a[m_1 : m_2],
3                     ...,
4                     map g a[m_{p−1} : m_p]]
5          ≡ flatten (map (map g)
6              [a[0 : m_1], a[m_1 : m_2], ..., a[m_{p−1} : m_p]])
7          ≡ let ys = map (map g)
8                [a[0 : m_1], a[m_1 : m_2], ..., a[m_{p−1} : m_p]])
9            in flatten ys
10         ≡ let (xs, ys) =
11               unzip <| map (λ_ _ a' → (0, map g a'))
12                  [m_1, m_2, ..., m_p]
13                  [m_1, m_2 − m_1, ..., m_p − m_{p−1}]
14                  [a[0 : m_1], a[m_1 : m_2], ..., a[m_{p−1} : m_p]])
15           in (reduce (λ_ _ → 0) 0 xs, flatten ys).2
16         ≡ (husk (λ_ _ → 0) 0
17                 (λ_ _ a' → (0, map g a')) a).2
```

Listing 8 A `reduce` operation using an array $\texttt{b} \in [m]\alpha$ and a monoid $(\oplus, \epsilon)$ where $\oplus : \alpha \to \alpha \to \alpha$ is an associative operator and $\epsilon \in \alpha$.

```
1 reduce (⊕) ϵ b
```

Listing 9 Rewriting of the `reduce` operation in Listing 8.

$$
\begin{aligned}
&\texttt{reduce}\ (\oplus)\ \epsilon\ \texttt{b} \equiv \epsilon \oplus \texttt{b}[0] \oplus \texttt{b}[1] \oplus \ldots \oplus \texttt{b}[\texttt{m}-1] \\
&\qquad\qquad \equiv \epsilon \oplus (\texttt{b}[0] \oplus \texttt{b}[1] \oplus \ldots \oplus \texttt{b}[\texttt{m}_1 - 1]) \oplus \\
&\qquad\qquad\quad (\texttt{b}[\texttt{m}_1] \oplus \texttt{b}[\texttt{m}_1 + 1] \oplus \ldots \oplus \texttt{b}[\texttt{m}_2 - 1]) \oplus \\
&\qquad\qquad\quad \ldots \oplus \\
&\qquad\qquad\quad (\texttt{b}[\texttt{m}_{p-1}] \oplus \texttt{b}[\texttt{m}_{p-1} + 1] \oplus \ldots \oplus \texttt{b}[\texttt{m}_p - 1])
\end{aligned}
$$

in the pseudo code language, the value of the first part of the tuple resulting from the `map` as well as the result of the reduction are never used so an arbitrary monoid can be used to make the transformation to a husk operation. Notice that in this case we simply use a monoid $(\lambda\_\_ \to 0, 0)$ over the set $\{0\}$, but it could in principle be any monoid as the value will be discarded.

By this we have that a `map` operation can be transformed to a semantically equivalent husk by partitioning the input array, applying the `map` operation to the partitions, and lastly concatenating the results.

3.1.2 Reduce

The specified semantics of the husk operator builds upon the `reduce` SOAC, however the means to introduce a valid transformation from a reduction to a husk requires some realizations. Consider an array $\texttt{b} \in [m]\alpha$ and a monoid $(\oplus, \epsilon)$ where $\oplus : \alpha \to \alpha \to \alpha$ is an associative operator and $\epsilon \in \alpha$ is the neutral element of $\oplus$ over α, we can make a reduction on `b` using the monoid as shown in Listing 8. Recalling the semantics of the `reduce` SOAC in Table 2 and by the definition of a monoid we can equivalently rewrite the expression as shown in Listing 9 where m_i for $i \in [1, 2, \ldots, p]$ is the end index of a given partition of the input data, with $m_i < m_{i+1}$ and $m_p \equiv m$. Notice, that since ϵ is the neutral element of $\oplus$ over α, i.e. for any $x \in \alpha$ we have $\epsilon \odot x = x \odot \epsilon = x$, we can insert it anywhere in the expression. We can again rewrite as shown in Listing 10, illustrating that a reduction is semantically equivalent to a reduction over reductions of an arbitrary number of partitions of the input data. Again we use that a `map` with a univariate function can be rewritten as a `map` using trivariate function with two of the parameters unused in a map, to match the defined semantics of the husk operator. Also, as with the transformation of the `map` SOAC, we discard one of the values of the husk result, however this time we are interested in the reduced result so we discard the concatenated result.

Listing 10 Final rewriting of the `reduce` operation, relating it to the husk operator.

```
1  reduce (⊕) ϵ b
2    ≡ ϵ ⊕ reduce (⊕) ϵ b[0:m_1] ⊕
3          reduce (⊕) ϵ b[m_1:m_2] ⊕
4          ... ⊕
5          reduce (⊕) ϵ b[m_{p-1}:m_p]
6    ≡ reduce (⊕) ϵ [reduce (⊕) ϵ b[0:m_1],
7                    reduce (⊕) ϵ b[m_1:m_2],
8                    ...,
9                    reduce (⊕) ϵ b[m_{p-1}:m_p]]
10   ≡ reduce (⊕) ϵ (map (reduce (⊕) ϵ)
11       [b[0:m_1], b[m_1:m_2],..., b[m_{p-1}:m_p]])
12   ≡ let xs = map (reduce (⊕) ϵ)
13       [b[0:m_1], b[m_1:m_2],..., b[m_{p-1}:m_p]]
14     in reduce (⊕) ϵ xs
15   ≡ let (xs, ys) =
16       unzip <| map (λm' _ b' → (reduce (⊕) ϵ b',
17                                 replicate m' 0))
18           [m_1, m_2,..., m_p]
19           [m_1, m_2 − m_1,..., m_p − m_{p-1}]
20           [b[0:m_1], b[m_1:m_2],..., b[m_{p-1}:m_p]]
21     in (reduce (⊕) ϵ xs, flatten ys).1
22   ≡ (husk (⊕) ϵ (λm' _ b' → (reduce (⊕) ϵ b',
23                               replicate m' 0)) b).1
```

Likewise, the part of the intermediate result to be concatenated is simply a list of 0 with `m'` elements, as to adhere to the required function type signature.

Notice that neither of the transformations use the offset and size parameters in their body lambda function. However, these parameters are used by some of the optimizations on husk operators potentially applied throughout the compilation process and may potentially become useful to users if the husk operator becomes an operator in the Futhark language.

3.2 Implementation Strategy

We have now been introduced to the husk operator, but we have yet to see why it is useful. The primary purpose of the husk operator is to express the distribution of a problem through a lambda function applied to partitions of the input array, known as the *husk body*, the results of which are then either concatenated or reduced using a given monoid. This means that through the use of the husk operator the compiler could potentially distribute a problem between multiple GPUs by isolating executions of the husk body on each, e.g. for n GPUs the input array is split into n partitions and the body of the husk is then executed n times, each execution using different GPUs for any GPU-related operations.

A benefit of the isolated execution of the husk body for different GPUs is that the compiler does not need to discern between the different memory spaces of the GPUs as each execution is associated with a specific GPU. This essentially means that the notion of a GPU memory space inside the Futhark compiler may refer to different memory spaces depending on the context in which it is in. Therefore, the Futhark compiler can stay mostly oblivious to its new ability to use multiple GPUs.

However, in order to actually make the programs utilize multiple GPUs the backends of the Futhark compiler must be altered to reflect the abilities of this new operator. Since GPU operations are not limited to the body of husks, we introduce the notion of a *main device*, which refers to the GPU that will be used for any GPU-related operations outside the body of a husk. This introduces a centralized execution model where outside the husk all GPU memory blocks with relevant information must reside on the main device. This in turn means that when a husk is executed, each GPU must get the relevant data from the main device before execution and likewise the results must either be transferred to the main device or must be placed into host memory.

Though the concatenation of results is left for the backend to handle as it sees fit, the reduction is done on the host exclusively. This is primarily done to avoid the need to potentially have additional kernels that need to be launched specifically for the reduction, which is likely inefficient in most cases as not only will the results have to be transferred to the main device from each other device, the reduction is often over scalars so with the typically few GPUs in a modern computer system the GPU doing the reduction over the result would very likely be under-utilized.

Part II
Husk Operator Implementation

4 Futhark Compiler Optimizer

To be able to identify the parts of the Futhark compiler that the husk operator will affect, we first need to gain an understanding of the stages in the compiler. The Futhark compiler can be separated into three stages, namely a frontend, an optimizer, and a backend, each producing the input for the next stage in the given order. The frontend is given the raw Futhark code of the program to be compiled and first lexes and parses it, the result of which is then type-checked and converted into an internal core representation of SOACs. The optimizer then takes the representation of the program generated by the frontend and applies various transformations to it. To allow for different transformations, multiple different optimizers exist in the Futhark compiler, known as pipelines. In the current state of the compiler there are two primarily used pipelines; the sequential CPU pipeline and the GPU pipeline, where both have overlapping transformations known as the common pipeline. As the names suggest, the sequential CPU pipeline is intended to run sequentially on a single thread of the CPU, whereas the GPU pipeline transforms the program to utilize the parallelism it offers, as to best use the resources supplied by a GPU. The result of passing the representation of the program from the frontend through the pipeline is then further transformed into an imperative representation of the program, known as *ImpCode*, which is then in turn passed to the corresponding backend. Backends can vary in how they work, but at the time of writing all backends transform the ImpCode representation into a program in another programming language, which is then compiled using a compiler for that particular language. ImpCode is currently the only intermediate representation used between the optimizer and the backends as all current backends use imperative languages. Notice that, in order to ensure that the transformations applied in the optimizer do not introduce type errors into the program, each result of a transformation inside the optimizer is type-checked.

Since the intention of husks are to utilize the resource of multiple GPUs, we will focus on the GPU pipeline. Figure 2 illustrates the operation of the Futhark compiler using the GPU pipeline with the frontend, optimizer, and backend, each with their corresponding steps. Notice that, since the simplification pass and the common sub-expression pass is applied often they have been replaced by a symbol and are applied in the order they are shown between other steps of the optimizer, whereas the type-checking between each step is assumed implicit and therefore omitted for simplicity.

In relation to the compiler passes shown in Figure 2, since the husk operator is internal to the compiler there is no need to make changes to the frontend. Instead, husks are generated in the kernel extraction step of the optimizer stage and is therefore first represented in the core language using kernels. In this section we explore the creation and transformations of husks in the Futhark compiler ahead of reaching the backend, which is in turn the focus of Section 5.

However, in order to express the transformations from SOACs to the husk operator, as described in Section 3.1, we first need to introduce some of the constructs of the intermediate SOAC representation passed to the kernel extrac-

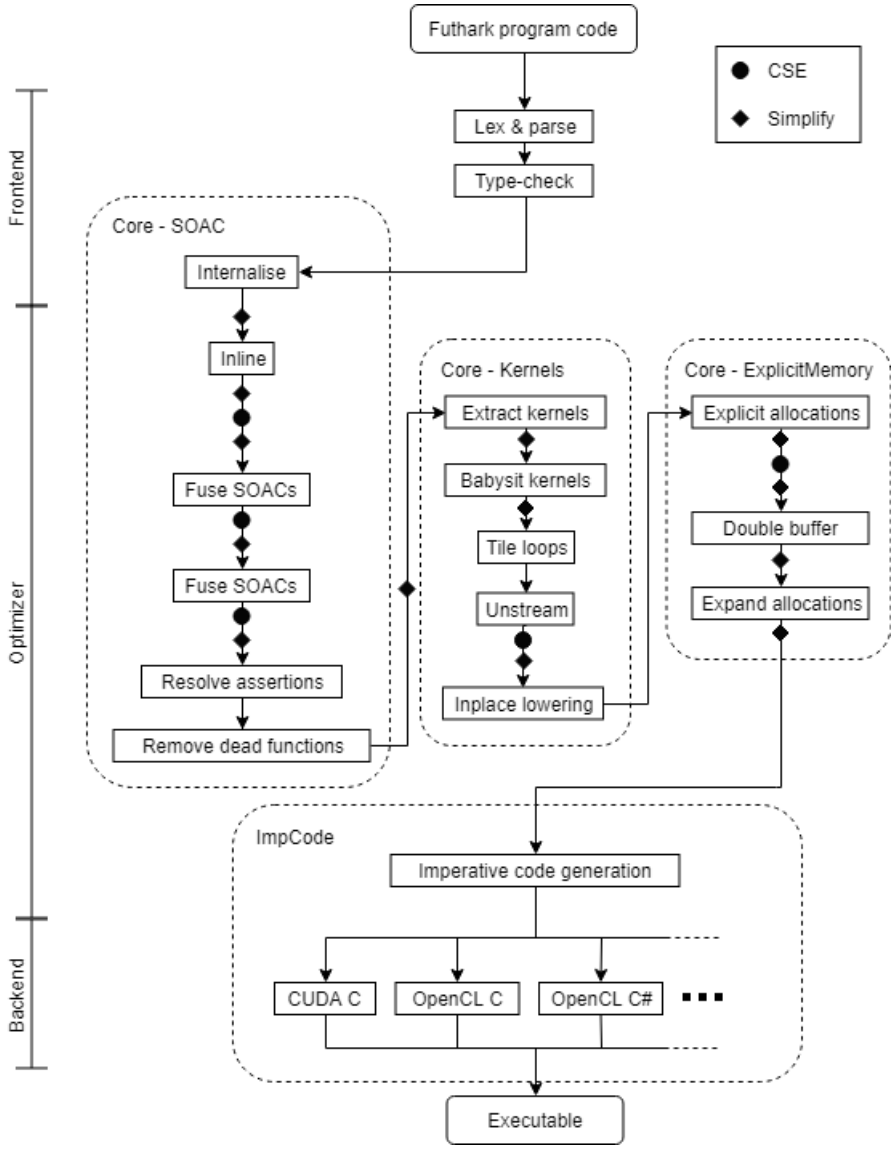

Figure 2: The compilation of a program using the GPU pipeline of the Futhark compiler. The primary stages of the compilation is separated into three steps; the frontend, the optimizer, and the backend. The stippled round-corned boxes specify the representation of the program after any of the steps it encapsulates. Notice that the type-checking between transformations in the optimizer has been omitted.

Listing 11 A high-level representation of the internal map SOAC.

```
1 soac_map
2   sm_m_lam
3   [sm_m] sm_arr
```

Listing 12 A Futhark program with a simple `map` operation.

```
1 let main [xs_m] (xs: [xs_m]i32): [xs_m]i32 =
2     map (+1) xs
```

tion step. Let L_{SOAC} be the set of all expressions in the intermediate SOAC representation language. Of particular interest are the map and redomap representations.

The first of these, the map SOAC representation is very similar to the corresponding Futhark operator. We introduce a high-level representation of the map SOAC as shown in Listing 11 where the attributes are as follows:

`sm_m_lam` A lambda function to apply to each element of the input array, where the body is in L_{SOAC}.

`sm_arr` The name of the input array.

`sm_m` The outer size of the input array.

Likewise, the semantics of a `soac_map` construct is simply applying the map lambda function to each element of the input array. A `soac_map` is generated through the use of the Futhark `map` SOAC, so a simple example of it is the Futhark program in Listing 12 which adds 1 to each element of the input array `xs`. When compiling this program, just before reaching the kernel extraction pass, the map operation is represented as shown in Listing 13 .

Whereas the internal map SOAC is very similar to the `map` operator of the Futhark language, the internal redomap SOAC is not directly related to an operation. At high-level we can represent the internal redomap SOAC as shown in Listing 14 where the attributes are as follows:

`sr_m_lam` A lambda function to apply to each element of the input array, where the body is in L_{SOAC}.

`sr_r_lam` A lambda function for the reduction of the results of applying `sr_m_lam` to each element of the input array, where the body is in L_{SOAC}.

`sr_r_ne` The neutral element of `sr_r_lam`.

`sr_arr` The name of the input array.

`sr_m` The outer size of the input array.

Listing 13 Internal map SOAC representation of the `map` operation in the program in Listing 12.

```
1 soac_map (λx → x + 1) [xs_m] xs
```

Listing 14 A high-level representation of the internal redomap SOAC.

```
1 soac_redomap
2   sr_m_lam
3   (sr_r_lam, sr_r_ne)
4   [sr_m] sr_arr
```

The internal redomap SOAC represents a map-reduce composition with the map lambda function applied to each element of the input array, the result of which is then reduced using the (`sr_r_lam`, `sr_r_ne`) monoid. The internal redomap SOAC representation is also slightly more diverse than the internal map SOAC represenation as it is used to represent both reductions and map-reduce compositions. Consider the Futhark program shown in Listing 15 which, using a single reduce operation, sums all elements in the input array `ys`. Compiling this program results in the SOAC representation shown in Listing 16 before reaching the kernel extraction pass. Notice that, since there is no `map` operation involved, the map lambda function is the identity function, i.e. the input and the output are the same. Now consider the program in Listing 17 where we see a map-reduce composition with a `map` operation first adding 1 to each element of the input array `zs`, like the program in Listing 13, the output of which is then summed using a `reduce` operation, like in the program in Listing 15. Compiling this program results in the SOAC representation shown in Listing 18.

4.1 Kernel Extraction

In order to perform the SOAC transformations described in Section 3.1 we need an internal representation of the husk operator in the internal kernel representation language of the Futhark compiler. Let L_{kernel} be the set of all expressions representable in the intermediate kernel representation language, wherein the kernel husk representation is defined by the construct `kernel_husk` shown in Listing 19 where the internal parts are as follows:

`k_src` The name of the source array to be partitioned between nodes.

`k_m` The number of elements in the source array.

`k_p_arr` A name for the partition array.

`k_p_o` A name for a variable that represents the outer-dimension offset of the partition in the source array.

Listing 15 A Futhark program with a simple `reduce` operation.

```
1 let main [ys_m] (ys: [ys_m]i32): i32 =
2     reduce (+) 0 ys
```

Listing 16 Internal redomap SOAC representation of the `reduce` operation in the program in Listing 15.

```
1 soac_redomap (λy → y) (λy1 y2 → y1 + y2, 0) [ys_m]ys
```

Listing 17 A Futhark program containing a map-reduce composition.

```
1 let main [zs_m] (zs: [zs_m]i32): i32 =
2     reduce (+) 0 <| map (+1) zs
```

Listing 18 Internal redomap SOAC representation of the map-reduce composition in the Futhark program shown in Listing 17.

```
1 soac_redomap (λy → z + 1) (λz1 z2 → z1 + z2, 0) [zs_m]zs
```

Listing 19 A high-level representation of a husk in the intermediate representation language of kernels.

```
1 kernel_husk
2   [k_m]k_src
3   (k_p_arr, k_p_o, k_p_m)
4   {k_b_exp}
5   (k_r_lam, k_r_ne)
```

`k_p_m` A name for a variable that represents the outer size of the partition.

`k_b_exp` An expression in L_{kernel}. This will be referred to as the body of a kernel husk.

`k_r_lam` A lambda function for the reduction of the husk body results, where the body is in L_{kernel}. This will be referred to as the kernel husk reduction.

`k_r_ne` The neutral element of the reduction lambda function `k_r_lam`.

As a generalization we allow the reduction lambda function to be a nil-function, i.e. a function with no input and no output, and correspondingly we allow the absence of its neutral element, in which case the results are concatenated. Throughout this section we will use `NilFn` and `Nil` to represent the nil-function and non-existing attribute value respectively. Notice that this is a restriction of this representation compared to the semantics of the husk operator in Listing 3 which is made in order to simplify the transformations. However, though the actual representation inside the compiler uses a very similar approach for distinguishing between a reduced result and a concatenated result, it is actually more permissive as it allows any number of reduced results and any number of combined results for a single husk operator.

The semantics of the kernel husk are illustrated in Figure 3 which shows that if we have d nodes, which are in our case d GPUs, the source array `k_src` is separated into d partitions each with a partition of size m_i at offset o_i for $i \in [1, \ldots, d]$ which are all under the name of `k_p_arr` on the nodes. The variables representing the offset and the partition size, `k_p_o` and `k_p_m` respectively, are initialized before running the husk body on each device. The result of running the husk body on each device are then combined by use of the husk reduction lambda or concatenated if the reduction function is absent. Notice that in this illustration the results are stored in the array `node_res` which is not actually a part of the kernel husk representation but is introduced here for illustration purposes. The actual handling of the body results is left to the backend. Likewise, the partition sizes and offsets are to be specified by the compiler backend being used by the end of the compilation of the program, so we do not necessarily have that o_{i+1} is $o_i + m_i$. However, the operator for the husk reduction may not be commutative, so the husk reduction must know the order of the partitions to infer the order of the results as to compute the husk reduction correctly.

With the kernel husk representation we can generalize the SOACs transformations as the following steps:

1. Create a name for the partition of the input array and variable names for the partition's element offset into the input array and the number of elements in the partition.

2. Let the input array be the source array of the husk.

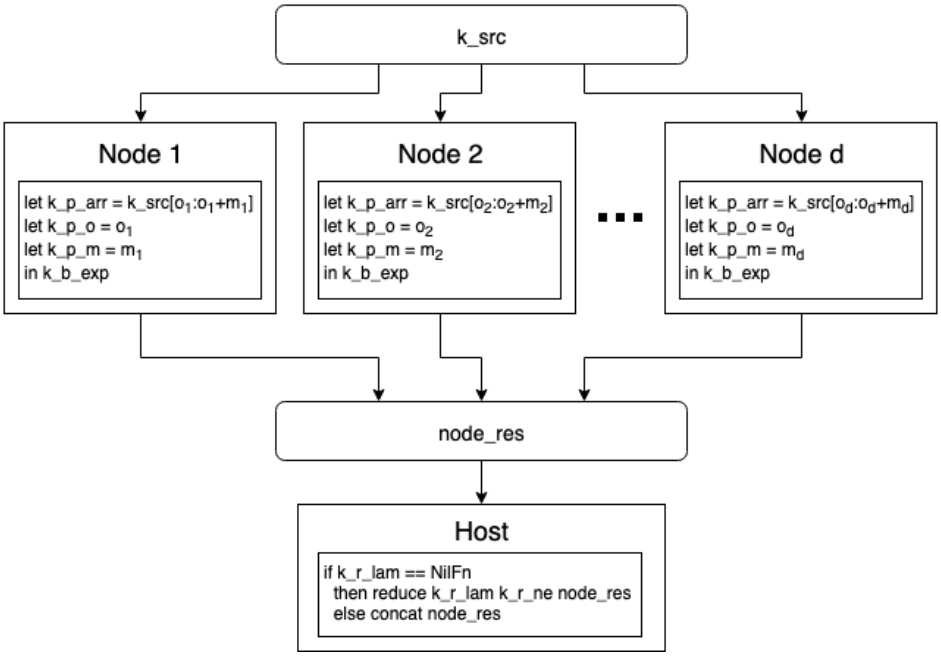

Figure 3: Illustration of the semantics of a kernel husk inside the Futhark compiler.

Listing 20 The transformation of an internal map SOAC to a kernel husk.

```
1  E(soac_map sm_m_lam [sm_m] sm_arr) ≡
2    kernel_husk
3      [sm_m] sm_arr
4      (arr', o', m')
5      {E'(soac_map sm_m_lam [m'] arr')}
6      (NilFn, Nil)
```

Listing 21 A kernel husk transformed from the internal map SOAC shown in Listing 13.

```
1  kernel_husk
2    [xs_m] xs
3    (arr', o', m')
4    {E'(soac_map (λx → x + 1) [m'] arr')}
5    (NilFn, Nil)
```

3. Generate a monoid for the reduction of the intermediate node results. For redomaps this is simply the corresponding reduction operator and neutral element, whereas maps have a nil-function.

4. Transform the SOAC as normal, but on the partition of the input array instead of the input array.

In order to express the transformations in terms of the intermediate SOAC representations and the intermediate kernel representations, we need to introduce the transformation functions $E : L_{SOAC} \to L_{kernel}$ and $E' : L_{SOAC} \to L_{kernel}$. Let $E(s)$ be the kernel representation of the internal SOAC representation s outside of husk bodies and let $E'(s)$ be the kernel representation of the internal SOAC representation s inside the body of a husk. Likewise, we have a variant of E', named E'_{lam} that transforms a lambda function with a body in L_{SOAC} to a lambda with a body in L_{kernel}. With these we can describe the kernel extractions of a kernel husk from an internal map SOAC as shown in Listing 20 where we use the same name for the internal attributes as in the definition in Listing 11, but let `arr'`, `o'`, and `m'` be generated names. As described in Section 3.1, this illustrates how a map is converted to a husk with a map applied to each partition. As an example, consider the internal map SOAC in Listing 13, which would result in the kernel husk representation shown in Listing 21 where again `arr'`, `o'`, and `m'` are generated names. Notice that the transformation of the map SOAC representation uses the special case of the kernel husk representation where the nil-function as the husk reduction lambda function results in the intermediate map results being concatenated, as required by the transformation of map.

Likewise, the redomap SOAC representation is transformed as shown in List-

Listing 22 The transformation of an internal redomap SOAC to a kernel husk.

```
1  E(soac_redomap sr_m_lam (sr_r_lam, sr_r_ne)
2      [sr_m] sr_arr) =
3    kernel_husk
4      [sr_m] sr_arr
5      (arr', mem', o', m')
6      {E'(soac_redomap sr_m_lam (sr_r_lam, sr_r_ne)
7           [m'] arr')}
8      (E'_lam(λz1 z2 → z1 + z2), 0)
```

Listing 23 A kernel husk transformed from the internal redomap SOAC shown in Listing 18.

```
1  kernel_husk
2    ys [ys_m]
3    (arr', o', m')
4    {E'(soac_redomap (λy → y) (λy1 y2 → y1 + y2, 0)
5         arr' [m'])}
6    (E'_lam(λy1 y2 → y1 + y2), 0)
```

ing 22 where we use the same name for the internal parts as in the definition in Listing 14, but let `arr'`, `o'`, and `m'` be generated names. This means that transforming the intermediate SOAC representation in Listing 16 results in the kernel husk shown in Listing 23 whereas the intermediate SOAC representation in Listing 18 is transformed into the kernel husk shown in Listing 24 .

Notice the subtle difference between E and E', namely that the first is the transformation of an expression in the SOAC representation outside a husk body, whereas the latter is a transformation of an expression in the SOAC representation within the body of a husk. Since we are only going to need a single level of partitioning in order to distribute kernels between multiple GPUs, additional husk levels would only complicate further compilation. Additionally, even if multiple nested husks were allowed, using E would result in an infinite nesting of husks which would stall compilation. Given the generality of the husk construct (see Section 9.1.1), only allowing a single level of husks should also be enough to allow hierarchical multi-node systems as well. This single level of husks does however assume that the outer parallelism of the contained operation is sufficient to utilize all resources available. If this assumption does not hold, further distribution of the inner parallelism may be beneficial, but would require additional analysis at runtime.

Some generated redomap SOACs may also return the result of the map part of its computation together with the reduction result. Take for example the Futhark program in Listing 25 where the result `xs'` of the map, which is in turn part of the indirect map-reduce composition resulting in `xs"`, is used outside

Listing 24 A kernel husk transformed from the internal map SOAC shown in Listing 18.

```
1  kernel_husk
2    [zs_m]zs
3    (arr', o', m')
4    {E'(soac_redomap (λz → z + 1) (λz1 z2 → z1 + z2, 0)
5         [m']arr')}
6    (E'_lam(λz1 z2 → z1 + z2), 0)
```

Listing 25 A Futhark program that produces a redomap composite kernel in the GPU pipeline of the Futhark compiler, but where the result of the map is used afterwards.

```
1  let main(xs: []i32): ([]i32, i32) =
2      let xs' = map (+1) xs
3      let xs'' = reduce (+) 0 xs'
4      in (xs', xs'')
```

the map-reduce composition, specifically as the part of the composite result of the program. As given, our definition of the intermediate redomap SOAC representation does not allow this as it uses only a single source array and a single output. However, this is just an implementation detail of the Futhark compiler as the program could in principle be represented as an internal map SOAC with the result `xs'`, followed by an internal redomap SOAC with the result `xs"`. As another implementation detail of the Futhark compiler, both SOAC representations may actually have multiple input arrays and multiple outputs, though this is not representable in the language we established here for exposition purposes. In general such SOAC representations will still have a single map lambda function and a single reduce lambda function, each with the number of arguments corresponding to the number of input arrays. Likewise the neutral element becomes a list of neutral elements, in effect acting as a composite neutral element of the combined reduce lambda function. The husk representation introduced here is in the actual implementation adjusted to take a number of source arrays, have the same number of partitions, and have multiple outputs. In order to handle the special case where parts of the output may be a map result, the reduction lambda function will only have a number of input arguments corresponding to the number of output values of the map lambda function that should not just be concatenated, and likewise the neutral elements list will only have the corresponding number of neutral elements. Notice that a restriction of both the internal map and redomap SOACs in these cases are that all input arrays must have the same number of elements, thus a husk needs only a single variable representing partition offsets and the number of elements in the partitions. Consequently, the concatenated outputs of the SOACs will

also have the same number of elements in the output arrays.

4.2 Simplification

The simplification pass of the Futhark compiler attempts to apply a set of simplification rules to the intermediate representation of the program being compiled until convergence is reached. These rules are various, but of note is that expressions nested inside the body of another expression, for example inside the body of a loop, may be moved outside of it, which may in turn allow for more simplifications. Since husks have variables that are defined only in the body of the husk itself, namely the partition, its offset into the source array, and the number of elements in the partition, these must be blocked from leaving the body of the husk.

Throughout the transformations and simplifications of a program some variables may become obsolete, e.g. if all the expressions using them have been transformed to an expression that does not use them. In the case that a variable is no longer used the simplifier will remove it. In general this will be carried out on the husk body and reduction operator simply by use of the existing simplification rules on bodies. However, since the source arrays of husks are implicitly used by the corresponding husk in order to create partitions, it must specify that the source arrays are used by the husk in order prevent the simplifier from wrongfully removing them.

Though the husk reduction and body are simplified using the normal rules for their respective representations, making sure the specified simplification restrictions are upheld, the husk carries additional information, namely the source array, the partition, and information about the partition, which must also be reachable by a simplification pass in order to allow change to propagate to them, such as for example changing the name of the source array if it was an alias for another array.

4.2.1 Husk-specific Simplification Rules

Though there are the general simplification rules for removing dead variables and hoisting expressions, a subset of the applied simplification rules are more specialized, targeting specific constructs inside the representation. In line with this, a number of husk-specific rules have been defined primarily for use on husks in the intermediate kernel representation. Though more have been introduced we will focus on two as to not get to far into minor implementation details.

The first simplification rule we introduce is the removal of unused partitions. This simplification rule checks that the partition of the source array is used inside the body of the husk. Since partitions are only defined inside the husk body the absence of it means that the partition is completely unused and can therefore be omitted in its entirety. By removing the unused partitions we can prevent expensive inter-GPU copying of unused data. For example, consider the Futhark program in Listing 26 in which the simplifier will remove the use of the parameter `x` inside the lambda function. The husk representation of this

Listing 26 A Futhark program where the simplifier will remove any uses of `x` inside the application of `map`.

```
1 let main [m] (xs: [m]i32): [m]i32 =
2   map (\x -> x * 0) xs
```

Listing 27 The husk representation in Listing 29 with its partition having been inserted into the body and removed from the construct itself.

```
1 kernel_husk
2   [m]xs (Nil, o', m')
3   {
4     ...
5   }
6   (NilFn, Nil)
```

program would be as shown in Listing 27 where the partition would no longer be used inside the body of the husk as it would be unused by the map and will therefore have been removed by the simplifier. Notice that this does in principle require our definition of husks in the internal kernel representation to be changed in order to allow the absence of a partition, but in order to keep it simple we will keep it in the construct. However, as with having multiple partitions in a husk, having no partitions is a simple generalization that can fairly easily be applied to the husk representations.

The other simplification rule of interest is the internalization of partitions, i.e. moving the partition inside the body of the husk as a slice of the source array if the source array is itself used inside the body of the husk. For example, consider the Futhark program in Listing 28 which will result in a kernel representation of the program containing the husk as shown in Listing 29. Since one of the key ideas of having partitions is to reduce the amount of memory that need to be transferred to the devices, having the source array in the body of the husk would mean that the source array would have to be broadcast to all devices and therefore the partitioning of the source array would be redundant and unnecessarily expensive. Therefore, if the source array of the partition appears inside the body of the corresponding husk the partition can simply be defined as a slice of the array inside the body. With this the husk representation in Listing 29 is transformed into the kernel representation of a husk shown in Listing 30 where the partition has been inserted into the body of the husk as a slice on line 4 and has in turn been removed as a partition of the husk construct itself on line 2. Notice that, like with the previous simplification rule, we assume that the partition is allowed to be absent from the construct which we will only allow in this section for the purpose of explaining the simplifications but will not be reflected in the following sections.

In both cases we cannot remove the offset `o'` and number of elements `m'` of

Listing 28 A Futhark program that will result in a husk where the source of the husks partition, i.e. `xs`, is used inside the husk body.

```
1 let main [m] (xs: [m]i32): [m]i32 =
2   map (\x -> x + unsafe xs[x]) xs
```

Listing 29 A husk in the internal kernel representation of the Futhark program in Listing 28.

```
1 kernel_husk
2   [m]xs (arr', o', m')
3   {
4     ...
5     let y = xs[x]
6     ...
7   }
8   (NilFn, Nil)
```

Listing 30 The husk representation in Listing 29 with its partition having been inserted into the body and removed from the construct itself.

```
1 kernel_husk
2   [m]xs (Nil, o', m')
3   {
4     let arr' = xs[o':o'+m']
5     ...
6     let y = xs[x]
7     ...
8   }
9   (NilFn, Nil)
```

Listing 31 A high-level representation of the husk construct in the intermediate explicit memory representation.

```
1 expmem_husk
2     [em_m] em_src
3     (em_p_arr, em_p_mem, em_p_o, em_p_m)
4     {em_b_exp}
5     (em_r_lam, em_r_ne)
```

the partition as the body of the husk may still depend on the partitioning logic determining the offset and size of the slice into the source array. Likewise, we could in principle remove the source array from the husk construct but since the size of it is likely to be used when determining `o'` and `m'` it is kept.

4.3 Explicit Allocation

The explicit allocation pass of the Futhark compiler analyses the intermediate kernel representation in order to find all needed memory allocations, transforming the representation to explicitly express these memory allocations. This introduces a new representation known as the intermediate explicit memory representation. Let L_{expmem} be the set of all expressions in the explicit memory representation language. With the addition of husks in the compiler we introduce a new construct in the intermediate explicit memory representation, namely the `expmem_husk`, as defined in Listing 31 where the attributes are as follows:

`em_src` The name of the source array to be partitioned.

`em_m` The number of elements in the source array.

`em_p_arr` A name for the partition array.

`em_p_mem` A name for the memory block for the partition array.

`em_p_o` A name for a variable that represents the outer-dimension offset of the partition in the source array.

`em_p_m` A name for a variable that represents the outer size of the partition array.

`em_b_exp` An expression in L_{expmem} representing the husk body.

`em_r_lam` A lambda function for the reduction of the husk body results. The body of this lambda function is in L_{expmem}.

`em_r_ne` The neutral element of the reduction lambda function `em_r_lam`.

From this representation the avid reader will have noticed that it is almost identical to the representation of the husk in the intermediate representation language of kernels but with the the expression of `em_b_exp` and `em_r_lam` being in the intermediate explicit memory representation language and the addition of the name for the memory block of the partition array `em_p_mem`. Likewise, the semantics of the of this construct is equivalent to that of the kernel husk representation with the addition of the access of partitions being done by use of the new memory block.

With the introduction of memory blocks we also have the introduction of index functions which represent the mapping from an index of the array to the corresponding element's location in the array's memory block. The simplest variant of an index function is the *direct index function*, which represents a row-major ordering of the elements of an array in the corresponding memory, meaning that the rows of the array is contiguous in memory. In contrast to a direct index function is its transpose, namely a column-major ordered array, where the columns of the array are contiguous in memory. Though index functions can be much more complex than these two examples they are enough to illustrate one of the key difficulties of the implementation of the husk operator, specifically that some source arrays are unsuitable for partitioning due to how they are represented in memory. As an example, consider the two-dimensional arrays `xss` and `yss` both with outer size of 4 and inner size of 3 but whereas `xss` is row-major ordered in memory, `yss` is column-major ordered. Consider now we partition these equally, which would mean the arrays are sliced in the outer dimension resulting in the partitions `xss'` and `xss"` of `xss`, as well as `yss'` and `yss"` of `yss`, all having an outer size of 2 and inner size of 3. An illustration of the memory layout of `xss`, `yss`, and their respective partitions, is shown in Figure 4 . We see how partitioning of the row-major array `xss` can be done by a single copy for each partition given the rows of the partition is contiguous in the memory of `xss`, thus requiring a total of 2 memory copies. However, for `yss` that has column-major ordering in memory we see that the rows of its partitions are not contiguous in its memory and thus needs 3 copies for each partition resulting in a total of 6 memory copies. Let p be the number of partitions and let a be a d-dimensional array with the dimension sizes d_i for $i \in [1, 2, \ldots, d]$. Notice that the the ordering is preserved with the illustrated partitioning. We can write the number of copies needed if the memory of a is row-major ordered as

$$min(p, d_1)$$

whereas if the memory of a is column-major ordered the number of copies needed are

$$min(p, d_1) \sum_{j=2}^{d} j$$

thus the number of copies needed for a column-major ordered array is a magnitude of the sum of the inner dimensions larger than a row-major ordered array

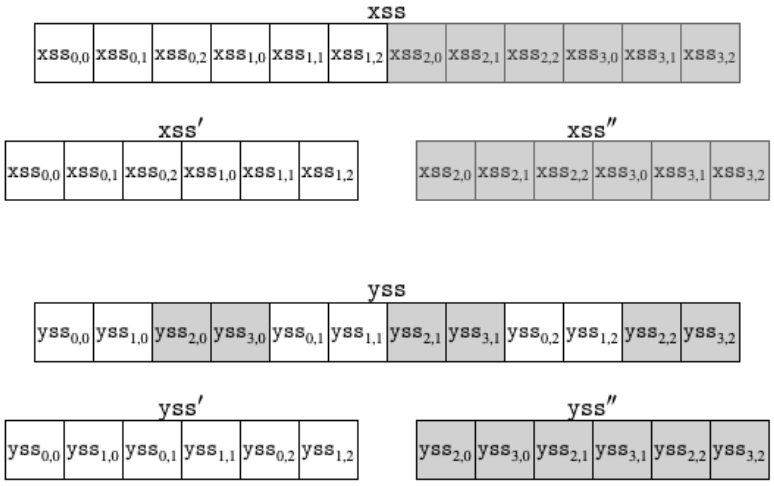

Figure 4: Illustration of the memory layout of the arrays `xss` and `yss`, with row-major ordering and column-major ordering respectively. The arrays `xss'` and `xss"` are even partitions of `xss` and the arrays `yss'` and `yss"` are even partitions of `yss`.

of the same size. Notice that an array cannot be split into more partitions than the outer size of the array, therefore giving the use of the minimum operator. Notice also that with more complex index functions the copy pattern needed to be able to partition correctly also generally becomes more complex.

Though the total amount memory that must be copied in order to perform the partitions stays the same no matter what index function is in use, the overhead from the intercommunication introduced by doing large amounts of small copies, e.g. as a column-major ordered array may require in order to be partitioned, may prove expensive. Instead of employing this complex and potentially expensive strategy for partitioning we only allow arrays with direct index functions to be partitions for which two alternative strategy were considered:

1. Check the index function of the source array and if it is not direct then restructure the memory of the array into a new memory block and a new array which will in turn have a direct index function. If the source array is restructured through this the resulting array becomes the new source array. For example, if the array is column-major ordered the array is be transposed. This strategy may require additional memory operations but these will be in the memory of the same device rather than between devices.

2. If the index function of the source array is not direct then the partition is inserted into the body of the husk as a slice of the source array and is not included as an explicit part of the husk construct. This will not require restructuring of the source array but will require the entire source array

to be copied to the other devices, increasing the amount of memory to be copied.

The performance of these strategies depend on multiple factors such as the bandwidth of the interconnect between devices and the bandwidth and speed of the memory on the main device. However, the second alternative strategy has a significant benefit, namely that it does not introduce additional overhead if there is only a single device which will be explored further in Section 6. In order to support this in the introduced construct `expmem_husk` we allow the source array `[em_m]em_src` and the partition tuple `(em_p_arr, em_p_mem, em_p_o, em_p_m)` to be the special `Nil` value representing the absence of a partition.

A similar problem occurs when the intermediate result of a husk to be concatenated do not have a direct index function. In principle this problem is also illustrated by Figure 4 but in this case we are copying smaller memory blocks into a larger memory block representing the result. Like when the source arrays of the partition do not have direct index functions we force the intermediate results to have direct functions but in this case we cannot employ a strategy similar the one chosen for partitions. Instead we employ a strategy similar to the first strategy considered for the partitions, namely that we ensure that the intermediate result arrays have a direct index function by restructuring them, thus the combined result of these results will also have a direct index function. Notice that this restructuring is done on each device and therefore on smaller memory blocks than it would have been if the same strategy was employed for partition, thus also potentially making it more viable for results than for partitions but still potentially introducing an overhead in case only a single device is used.

With the background established we can express the transformation from the intermediate kernel representation to the explicit memory representation by the transformation function $A : L_{kernel} \to L_{expmem}$, whereas we have the additional transformation function variant A_{lam} for the transformation for lambda functions. Additionally, let the trivariate function S be defined as

$$S(a, x, y) = \begin{cases} x & \text{The index function of } a \text{ is direct.} \\ y & \text{Otherwise} \end{cases}$$

for an array a, any x, and any y. The transformation of the `kernel_husk` construct of the intermediate kernel representation can be expressed as shown in Listing 32 where the names from the definition of the `kernel_husk` construct is reused and `mem'` is a generated unique name. The source array and partition tuple is only included as an attribute in the construct if the source array has a direct index function as ensured by the use of S in lines 7 and 8. However, lines 10-12 show that if the source array does not have a direct index function the partition array is inserted into the body as the slice of the source array. Notice that we use a Futhark-like notation for the expressions inside the body of the husk with `skip` representing a special no-operation expression that has no effect on the semantics of the program. We let these expressions be in L_{kernel} and

Listing 32 The transformation of a husk in the intermediate kernel representation to a husk in the intermediate explicit memory representation.

```
1  A(kernel_husk
2        [k_m] k_src
3        (k_p_arr, k_p_o, k_p_m)
4        {k_b_exp}
5        (k_r_lam, k_r_ne)) ≡
6    expmem_husk
7        S(k_src, [k_m] k_src, Nil)
8        S(k_src, (k_p_arr, mem', k_p_o, k_p_m), Nil)
9        {
10         (A ∘ S)(k_src,
11                 skip,
12                 let k_p_arr = k_src[k_p_o:k_p_o+k_p_m])
13         A(k_b_exp)
14       }
15       (A_lam(r_lam), k_r_ne)
```

transform using the transformation function A. Notice also that for simplicity the expression between the curly brackets, i.e. in lines 9-14, are combined in the order they occur.

As an example of this transformation, consider again the kernel representation in Listing 24 of the Futhark program in Listing 17. To illustrate the partitioning strategy we first consider the case where the source array `zs` has a direct index function which results in the explicit memory representation shown in Listing 33 . Notice the partition tuple and source array is represented as part of the constructs attributes in lines 1 and 2 as the source array index function is direct. For the same reason, at line 5 we have the additional `skip` operation which in principle can be omitted as it has no effect on the program and likewise the transformation of it has no effect either. Now consider the case where `zs` is not direct with which the transformation is instead as shown in Listing 34 where we notice that only the source array, the partition tuple, and the start of the body are different from the transformation result when the index function of `zs` is direct. Notice that unlike with the transformation in Listing 33 we cannot in this case omit the expression on line 5, i.e. the slice of `zs`. None of the transformation examples takes into account that there may have been made optimizations to the representation of their reduction function or their husk bodies. This is generally not the case but is done to allow for running examples through the optimizer transformations.

The need to keep the partition array and its memory block out of the body, making it implicitly declared and allocated, is atypical compared to other constructs in the explicit memory representation language. However, it is done in order to keep the generated representation oblivious to the fact that husks

Listing 33 The resulting explicit memory representation of the kernel husk shown in Listing 24 when applying the transformation function A, if `zs` has a direct index function.

```
1  expmem_husk
2    [zs_m] zs
3    (arr', mem', o', m')
4    {
5      A(skip)
6      (A ∘ E')(soac_redomap (λz → z + 1)
7                            (λz1 z2 → z1 + z2, 0)
8                            [m'] arr')
9    }
10   ((A_lam ∘ E'_lam)(λz1 z2 → z1 + z2), 0)
```

Listing 34 The resulting explicit memory representation of the kernel husk shown in Listing 24 when applying the transformation function A, if `zs` does not have a direct index function.

```
1  expmem_husk
2    Nil
3    Nil
4    {
5      A(let arr' = zs[o':o'+m])
6      (A ∘ E')(soac_redomap (λy → z + 1)
7                            (λz1 z2 → z1 + z2, 0)
8                            [m'] arr')
9    }
10   ((A_lam ∘ E'_lam)(λz1 z2 → z1 + z2), 0)
```

Listing 35 A high-level representation of a husk function in the ImpCode language.

```
1  imp_husk_function
2    hf_name
3    (hf_src, hf_part, hf_pbo, hf_pbs)
4    (hf_map_hres, hf_map_res, hf_mrbo, hf_mrbs)
5    hf_repl
6    hf_params
7    hf_o
8    hf_m
9    hf_src_m
10   hf_node_id
11   {hf_body_code}
```

introduces memory spaces on different devices that in turn need special operations to allow intercommunication. If we were to declare and allocate a partition inside the body of the husk then the copy from the memory block of the source array into the memory block for the partition, which is between devices and is therefore possibly different from the usual device memory copying, must also be inserted into the body but is currently not expressible in the representation language. Notice that any arrays and corresponding memory used, but not defined, inside the body of the husk must also be copied to the device that is running the given instance of the husk body, but since these are not partitioned they are simply a full copy and does not need to be identified until the imperative code representation.

4.4 Imperative Code Generation

With the GPU pipeline having done its transformations on the program representation the compiler has all the needed information to be able to continue to the backend. However, before reaching the backend the representation of the program is transformed into an intermediate representation that is closer to the imperative code that the backends will generate, which we will refer to as the ImpCode. With the introduction of the husk operator we need to introduce a transformation from its explicit memory representation to ImpCode, however in order to do so we first introduce a new construction we will refer to as a *husk function*. We define the husk function representation as shown in Listing 35 where the attributes are:

`hf_name` A unique name for the husk function.

`hf_src` A name for the source memory block.

`hf_part` A name for the memory block containing the partition of the source memory.

`hf_pbo` An ImpCode expression representing the byte offset of the partition into the source array.

`hf_pbs` An ImpCode expression representing the byte size of the partition.

`hf_map_hres` The name of the memory block containing the concatenated result of the husk function.

`hf_map_res` The name of the memory block for the concatenated results of the entire husk.

`hf_mrbo` An ImpCode expression representing the byte offset into the concatenated result where the result of the husk function is to reside.

`hf_mrbs` An ImpCode expression representing the byte size of the result of the husk function.

`hf_repl` A list of memory blocks that are used inside the body of the husk function.

`hf_params` A list of parameters for the husk function.

`hf_node_id` A name for the special parameter representing the unique identifier of the GPU that the husk function will use.

`hf_o` A name for the variable to contain the element-based offset of the partition into the source array.

`hf_m` A name for the variable to contain the outer size of the partition array.

`hf_src_m` An ImpCode expression specifying the number of elements in the source memory block.

`hf_body_code` An ImpCode representation of the body of the husk function.

Notice that we allow the case where there is no concatenated results which is represented by the concatenated result tuple containing `hf_map_hres`, `hf_map_res`, `hf_mrbo`, and `hf_mrbs`, being the special `Nil` value. Likewise, the partition tuple containing `hf_src`, `hf_part`, `hf_pbo`, and `hf_pbs` is also allowed to be `Nil`.

The execution of a husk function is the execution of the body `hf_body_code` with the partition `hf_part` having been copied from the source memory `hf_src` at the byte offset `hf_pbo` with a byte size of `hf_bps`. When the execution of the body is done, the map result `hf_map_hres`, with the byte size `hf_mrbs`, is copied to the concatenated result memory block of the entire husk `hf_map_res` at the byte offset `hf_mrbo` if they exist. In order to ensure correct execution of the body and of the partitioning the variables `hf_o` and `hf_m`, the parameters `hf_params,` and the special parameter `hf_node_id` must be in scope.

A husk function corresponds to a single execution of the distributed part of a husk and thus it is not a direct part of ImpCode but is instead a part of a husk construct in ImpCode. The ImpCode husk representation is specified as shown in Listing 36 where the attributes are:

Listing 36 A high-level representation of a husk in the ImpCode representation language.

```
1 imp_husk
2   ih_num_nodes
3   ih_hfunc
4   {ih_rres}
5   {ih_r}
```

`ih_num_nodes` A name for a variable to contain the number of GPUs to use for the husk.

`ih_hfunc` The husk function representing the distributed execution of the husk.

`ih_rres` An ImpCode representation of the allocation of a memory block on the host.

`ih_r` An ImpCode representation of the reduction of the reduction results of the husk function executions.

The execution of an ImpCode husk is done by execution of the husk function `ih_hfunc` with each GPU where the number of GPUs specified through the variable `ih_num_nodes`. The results are then reduced using `ih_r` which is done on the host. Notice that in order to do a reduction on the host, a memory block for the results from the body to be reduced is allocated on the host before executing the husk function and is given as the ImpCode piece `ih_rres`. Since there are a result for each GPU the amount of memory to allocate is equal to the number of GPUs, i.e. `ih_num_nodes`, times the size of the result type. Additionally, as the results may just need to be concatenated, which is in turn done directly to the main device, the allocation on host as well as the reduction ImpCode may be empty. Notice that since the husk function is executed for each GPU the host memory block for the results to be reduced cannot be allocated as part of the husk function and is therefore a separate attribute. Notice also that the allocations cannot be done outside the husk representation either as we must ensure that the allocation stays and is accessed as being on host, which is atypical for a construct generated through the GPU pipeline as discussed in Section 4.4.1.

Like with the previous transformations we again introduce a transformation function $I : L_{expmem} \rightarrow L_{impcode}$ where $L_{impcode}$ is the language of ImpCode expressions. In addition to the transformation function we need to define a number of auxiliary functions:

$mem(a)$ A function that takes an array a and returns the associated memory block.

$isize(m)$ A function that takes a memory block m and returns the total inner byte size of the corresponding array, i.e. the product of all other

dimensions than the first multiplied by the byte size of the element type.

$ismem(v)$ A logical function that returns $\top$ if the variable v refers to a memory block on the GPU and $\bot$ otherwise.

$free(e)$ A function that takes an expression e in L_{expmem} and returns the set of all free variables in the expression, i.e. all variables used but not defined inside the expression.

$lambda(l, x, y)$ A function that takes a bivariate lambda function l with a body in L_{expmem} and two expressions x and y in $L_{impcode}$, and returns an expression in $L_{impcode}$ representing l applied to x and y.

$rest(c, x, y)$ A function that takes arbitrary values c, x, and y, and returns x if c is the special Nil value and y otherwise.

Since ImpCode is an imperative representation we will use C-like pseudo code for any statically generated expressions of the transformation which is a better match to the structure of ImpCode than the Futhark-like pseudo code used in the previous representations. With this we can define the transformation of a husk in the explicit memory representation to an ImpCode expression as shown in Listing 37 where `name` is a unique name for the husk function and `num_nodes`, `node_id`, and `red_res` are unique variable names. Here we use `t` to denote the result type of the husk function and we use `res` to refer to the final result of the husk operation. As previously we use `skip` to represent an operation with no effect on the execution of the program.

The transformation introduced in Listing 37 may appear fairly involved so let us go through the different attributes of the resulting transformation:

Line 7 We introduce a unique name `num_nodes` for the auxiliary variable representing the number of executions of the husk function to be done and thereby also how many devices to use.

Lines 8-25 The husk function is generated in tandem with the creation of the husk construct. The attributes of the husk function are generated as:

Line 9 A unique name is created for the husk function. The exact name is not important and can in principle be generated arbitrarily as long as the uniqueness is ensured.

Lines 10-12 The partition tuple contains the memory corresponding to the source array, i.e. `em_src`, the name for the memory block of the partition `em_p_mem`, and the byte offset and size of the partition into the source respectively. Notice that the offset `em_p_o` and the size of the partition `em_p_m` are over the outer dimension of the source array and thus are have a byte-stride equal to the product of the inner size of the source array.

Listing 37 The transformation of a husk in the explicit memory representation to a husk in the ImpCode representation.

```
1  I(expmem_husk
2        [em_m] em_src
3        (em_p_arr, em_p_mem, em_p_o, em_p_m)
4        {em_b_exp}
5        (em_r_lam, em_r_ne)) ≡
6    imp_husk
7        num_nodes
8        (imp_husk_function
9          name
10         (mem(em_src), em_p_mem,
11          isize(em_p_mem) * em_p_o,
12          isize(em_p_mem) * em_p_m)
13         rest(em_r_ne,
14                (res, b_res,
15                 isize(b_res) * em_p_o,
16                 isize(b_res) * em_p_m),
17                Nil)
18         ({v ∈ free(em_b_exp) | ismem(v)} \ {em_p_mem})
19         (free(em_b_exp) \ {node_id, em_p_o, em_p_m, em_p_mem})
20         em_p_o em_p_m em_m
21         node_id
22         {
23           t b_res = I(em_b_exp)
24           rest(em_r_ne, skip, red_res[node_id] = b_res;)
25         })
26       {
27         rest(em_r_ne, skip,
28              t *red_res = malloc(sizeof(t)*num_nodes);)
29       }
30       {
31         rest(em_r_ne, skip,
32          {
33            t z = em_r_ne;
34            for (int i = 0; i < num_nodes; ++i){
35              z = lambda(em_r_lam, z, red_res[i])
36            }
37            res = z;
38          })
39       }
```

Lines 13-17 Since the husk function only contains the concatenated result tuple if there is no reduction associated with it, it is replaced by `Nil` if there is a neutral element specified. Notice that we could also have checked whether or not the reduction lambda function was the nil-function. The first element of the concatenated result tuple, if it is present, is the variable `res` representing the final result of the husk operation, since in this case it will be the concatenation of all intermediate results. The second element of the tuple is the variable `b_res`, i.e. the result of the body, whereas the last two elements are the byte offset and size respectively. Notice that if this tuple is present, the type `t` must be a memory block.

Line 18 The memory that must be broadcast to all devices is given as all free variables inside the body of the husk that refer to memory blocks on the GPU, except the partition memory block which is made specifically as to prevent the source array from being copied in full.

Line 19 Likewise, the parameters of the husk function are all the free variables in the body of the husk, except all variables declared implicitly by the husk, i.e. the auxiliary variables `node_id`, `em_p_o`, and `em_p_m`, as well as the partition memory block `em_p_mem`.

Line 20 The variable names representing the offset and size of the partition are propagated as they will be defined by the backend. Likewise we keep the outer size of the source array as it is likely to be relevant when declaring the partition offset and size.

Line 21 A unique name `node_id` is given to the special parameter that allows for the identification of the execution of the husk function ranging from 0 to `num_nodes` $-$ 1 which can in turn also be used to identify the corresponding device to be used by the husk function.

Lines 22-25 The body of the husk function corresponds to the transformation of the husk body in the explicit memory representation with its result written to some result variable `b_res`. If the results of the husk function calls are to be reduced then the result is written to the element of the array `red_res` corresponding to the execution of the husk function which can be identified by the husk function parameter `node_id`. Notice that since `red_res` is allocated on the host, as is done on lines 26-29, we assume an implicit device-to-host copy in the writing of the corresponding element in `red_res`.

Lines 26-29 If the results of the husk function calls are to be reduced then we must allocate a buffer for the individual results on the host which we do under a unique variable name `red_res` with a number of elements corresponding to the number of devices in use, which is identifiable through the variable `num_nodes`.

Lines 30-39 If the results of the husk function calls are to be reduced it accomplished by iterating over the results stored in `red_res` in order and apply

the transformed reduction lambda function to it with the final results written to `res`, i.e. being the final result of the entire husk operation.

An interesting note to this transformation is that, whereas the reduction is created here, the concatenation of results are left to the the backend. This is again to keep the optimizer oblivious to the multiple device memory spaces that the husk operator essentially introduces as otherwise the results would have to be expressed in terms of an inter-device memory copying operation from the results of each device to the main device which is what the backend will instead be in charge of carrying out by using the concatenated result tuple in lines 13-17. Notice that as it is written, this transformation does not take into account the case where the partition of the explicit memory representation of a husk is missing as is allowed due to the partitioning strategy discussed in Section 4.2.1. This was intentionally omitted as to prevent the transformation from becoming unnecessarily bloated but is as simple as having the partition tuple of the ImpCode husk be `Nil` if the partition is `Nil` in the explicit memory representation of the transformed husk.

Though the transformation shown in Listing 37 may appear confusing at first glance, much of it disappears once we can determine whether the husk function call results are to be reduced or concatenated. Consider as an example explicit memory representation of a husk in Lisiting 33 which would be transformed into a husk construct in ImpCode as shown in Listing 38 where `node_id'` and `num_nodes'` are unique variable names and `name'` is a unique husk function name. Additionally, for brevity we let $free'$ be the free variables in the body of the husk representation in Listing 33. We see here that since the results of the husk function calls are to be reduced there is no concatenation tuple in line 4 and the buffer allocation and reduction can be determined to be present.

4.4.1 Default Space

Within the GPU pipeline of the Futhark compiler whenever the program has been converted to ImpCode all memory blocks that are set to use the default space are changed to be device memory blocks. However, the husk representation needs the intermediate reduction result memory blocks to stay on the host which is normally denoted by the default space. Therefore, the tree-traversal function for setting the default spaces was changed to allow the exclusion of specific names from having their default space changed, keeping all access, allocation, and freeing in host memory. When setting the default space of a husk the list of intermediate reduction result memory blocks are then added to the exclusion list ensuring that they correctly stay on the host and is likewise correctly accessed both inside the husk function and when doing the reduction on the host.

Listing 38 The resulting ImpCode representation of the explicit memory husk representation shown in Listing 33 when applying the transformation function I.

```
1  imp_husk num_nodes'
2    (imp_husk_function name'
3      (mem(zs), mem', isize(mem') * o', isize(mem') * m')
4      Nil
5      ({v ∈ free' | ismem(v)} \ {mem'})
6      (free' \ {node_id', o', m', mem'})
7      o' m' zs_m node_id'
8      {
9        t b_res = (I ∘ A ∘ E')(
10         soac_redomap (λz → z + 1)
11                      (λz1 z2 → z1 + z2, 0)
12                      arr'[m'])
13       red_res[node_id] = b_res;
14     })
15   { t *red_res = malloc(sizeof(t)*num_nodes); }
16   {
17     t z = em_r_ne;
18     for (int i = 0; i < num_nodes; ++i){
19       z = lambda((A_lam ∘ E'_lam)(λz1 z2 → z1 + z2), z, red_res[i])
20     }
21     res = x;
22   }
```

5 Multi-GPU CUDA Backend

The Futhark compiler contains a number of GPU-centered backends primarily identified by the language of the generated code and the GPU compute API. Originally the only GPU compute API with a backend in the Futhark compiler was OpenCL using C for the host code whereas variants using Python and C# as the host language were introduced later on. Following this, the CUDA driver API and the Vulkan API have been used to make new backends both using C as the host language.

The introduced husk operator changes the structure of the generated intermediate representation of a Futhark program in the GPU pipeline thus requiring all Futhark backends utilizing the GPU pipeline to handle the case where husks are encountered. Due to the structure of husks, a simple implementation can be done by inserting the allocation, body, and reduction inline after setting the corresponding node identifier and number of nodes which in this case would be the constants 0 and 1 respectively. Such a backend is hardly interesting as it allows only for the execution of the husk on a single GPU, defeating the purpose of implementing the husk operator. However, as to prevent repetitive work for now we will leave most backends with this implementation and focus on a multi-GPU adjustment to a single one of the backends. Notice however that most decisions discussed in this section should be directly translatable to the other relevant backends.

For the remainder of this section we will focus on the CUDA C backend, which was chosen primarily because of the explicit nature of memory handling in the CUDA driver API. In contrast, the OpenCL API has more complex memory handling for multi-device systems, inferring device location implicitly, which can at times prove beneficial, but could potentially be inconvenient when attempting to adjust the backend from single-device to multi-device. The Vulkan API is possibly more explicit than the CUDA driver API in general but since the Vulkan backend for Futhark is still in development it was quickly excluded from this decision, lest it require debugging unrelated to the husk implementation. Notice however a drawback of focusing on the CUDA backend is that the results are limited to NVIDIA devices only.

In this section we introduce the changes needed to the existing structure of a C program generated through the Futhark CUDA C backend. To do this we first need to introduce the needed changes to the contexts of an execution of the program, that is the collection of information that is generally needed for the program to run such as a reference to the device and the memory management system. We then introduce the approach to generating C code from a husk function representation from ImpCode. In order to use a generated husk function we then introduce a worker-thread strategy for execution of a husk which utilizes the multi-threaded nature of modern CPUs to execute the host operations of a husk. Lastly we introduce the generating of C code that binds all the previous together, namely the launching of a husk.

For the rest of this section we let C denote a function from ImpCode to C code. Additionally, whenever we make an example for the generated C code any

Listing 39 The CUDA node context structure containing information on each node in the multi-node setting of the Futhark CUDA C backend.

```
1 struct cuda_node_context {
2   CUdevice dev;
3   CUcontext cu_ctx;
4   CUmodule module;
5
6   struct free_list free_list;
7 };
```

irrelevant information may be omitted and externally defined functions, such as C, may be used inline, thus any such examples should be seen as being written in a C-like pseudo language.

5.1 Multiple CUDA Contexts

In order for the CUDA driver API to be able to identify the device to be addressed by operations it exposes, a CUDA device context must be created. Likewise, if a program is to use multiple devices it must also create at least one context for each of these device. Though these contexts are the corner stone of most operations they are, more often than not, excluded from the parameters of the CUDA driver API operations. For these operations to identify the context to operate on, and thereby the device, the CUDA driver API maintains a thread-local context stack where the active context is at the top of the stack. Whenever a context is explicitly created, the created context is pushed onto the top of the stack.

Akin to the contexts, to be able to run a CUDA program on a GPU in a CUDA context a module must be created. Modules are collections of launchable CUDA kernels loaded from PTX programs which are in turn referenced by querying the API for the kernels by name. As each device needs the CUDA kernels of the program in order to be able to launch them, a module must be loaded through each device context that may use it.

In order for the Futhark CUDA backend to reflect this we introduce a new structure to the boilerplate C code, namely the node contexts, defined as shown in Listing 39 where we notice that the node contexts contains both a device reference in line 2, a CUDA device context in line 3, and a CUDA module in line 4. Notice that since each device has individual memory spaces the structure also has a free-list, i.e. the memory management construction of the generated runtime, on line 6 which is exclusive to the context specified. All allocation and freeing for the CUDA device context must be done using this free list, as they will be device specific. This new structure replaces the same variables in the CUDA context structure as an array with n elements where n is the number of GPUs to use upon execution.

To create node contexts the Futhark CUDA backend must find n suitable devices which are selected using the follow two filtering rules:

1. First find all GPUs with the highest CUDA API version amongst all of them, which generally are the newest GPUs in the system. This is done by sorting by the API version and truncating after the last with the highest API version.

2. Then enforce that these devices are symmetric which is determined by them having the same name. This is not strictly necessary but will make partitioning simpler, as we will discuss later.

This is a heuristic for finding the most suitable devices but highest API version does not necessarily mean that it is a better device. The first of the filter rules can be overridden using the Futhark CUDA backend command line parameter for specifying a preferred device, selecting only devices matching the name given. The rule of symmetry in devices is always enforced however.

When devices are determined we first generate the necessary PTX representation of the program, which is the intermediate representation of CUDA programs either compiled from a CUDA C representation of the program or read from a file. For each device we then create a new CUDA device context which combined with the PTX program is used to create a module. Notice that given the symmetry of the selected devices, and therefore also the same API version, the PTX program need only be generated once and can thus be used for all devices when loading a module, whereas with different API versions there may be different optimizations done to the programs and thus may benefit from different PTX programs.

When the module has been loaded the program extracts references to all the kernels. Since this reference are unique to each device we again need to define a new structure, namely the Futhark node context. However, the content of this structure is dynamic as the kernels involved depends on the program being compiled. Like with the CUDA node context the program needs to initialize one Futhark node context for each node to be executed on.

The number of GPUs to use, and thereby also the number of node contexts to create, is specified through the `nodes` command line argument of a Futhark program compiled using the CUDA C backend. As an example, for a Futhark program compiled to a file `prog` using the CUDA C backend, that can execute it with 2 GPUs using the command

```
1 > ./prog --nodes=2
```

where the number of nodes to use defaults to 1 if unspecified, whereas specifying the number of nodes to use as 0 will cause the runtime to use as many devices as possible fitting the specified criteria. This sets the `num_nodes` variable of the configuration inside the CUDA context to the value specified.

Introducing additional device memory spaces also introduces the need to be able to identify the correct node context in order to allocate and free not only as to have the correct free list but also to make sure that we are in the correct

Listing 40 An example of a husk function of a husk in ImpCode.

```
1  imp_husk_function
2    name
3    (src, part, pbo, pbs)
4    (hres, res, mrbo, mrbs)
5    [x]
6    [(src, memblock_device), (res, memblock_device),
7     (x, memblock_device), (y, t)]
8    po
9    pm
10   sm
11   node_id
12   {body_code}
```

CUDA device context when the allocation or freeing is done. To make the C code generator context aware we introduce a new element to the state structure, namely the local node identifier. The local node identifier is an expression containing the index of the currently relevant node contexts, i.e. the node identifier. This local node context is by default 0 as to specify the first node as the main node, whereas whenever the body of a husk is compiled the local node identifier is set to the expression identifying the relevant node for each iteration of the body and reduction operator. The local node identifier is then returned to the previous value upon completion of the husk compilation.

5.2 Generating Husk Functions

With the compilation of a husk inside the CUDA C backend we first generate the code for the corresponding husk function. To illustrate this code generation step, consider the husk function from an ImpCode husk shown in Listing 40 where we denote a list by a comma-separated elements between square-brackets, as shown for the memory blocks that need to be copied in line 5 and the parameters in lines 6-7. Additionally, we represent the parameters as a list of tuples where the first element is the name of the variable to bring into scope and the second variable is the type. Notice that `memblock_device` is a structure with properties `mem` and `size` representing a memory block on a device and its size respectively. The type `t` is an arbitrary C type.

By a design choice that will become apparent in the following sections, a husk function is compiled from ImpCode by generating a C function for it. The name of this function is the name of the husk function whereas the parameters are:

1. The full Futhark context of the execution of the program.
2. The node identifier, the name of which will be the variable name specified by the husk function. This is used to find the CUDA node context with the

Listing 41 The parameter structure husk_context generated from the husk function in Listing 40.

```
1  struct husk_context {
2    memblock_device src;
3    memblock_device res;
4    memblock_device x;
5    t y;
6  };
```

Listing 42 The parameter structure `husk_context` generated from the husk function in Listing 40.

```
1  int name(void *vctx,
2           int32_t node_id,
3           void *husk_params_p) {
4    futhark_context *ctx = (futhark_context*)vctx;
5    husk_context husk_params =
6      *(husk_context*)husk_params_p;
7
8    ...
9
10   return 0;
11 }
```

device corresponding to the execution of this husk function, e.g. through by use in the local node identifier.

3. A collection of values corresponding to the variables specified in the parameter list.

Notice that in order to generalize the signature of the generated function we combine all the parameters specified by the husk function into a collective structure. This will allow the backend to pass the parameters around without having to make special cases for each husk function in the program, the importance of which will become more apparent when the husk functions are to be used. This means that in addition to declaring a function for the husk function we also must generate a structure to contain all the parameters of the husk function. For the example of a husk function in Listing 40, the backend generates a parameter structure `husk_context` as shown in Listing 41 . With this structure the C function for the husk function is defined as shown in Listing 42 where we see further generalization of the Futhark context and the husk function parameters by use of the void pointer type that is in turn cast to the Futhark context structure and the generated parameter structure immediately inside the body of the function. This is again to generalize the type signature of the generated

Listing 43 The declaration of variables from the parameter structure inside the body of the function generated from the husk function in Listing 40.

```
1 memblock_device src = husk_params.src;
2 memblock_device res = husk_params.res;
3 memblock_device x;
4 t y = husk_params.y;
```

Listing 44 The copying of `x` from the main device to the corresponding device inside the function generated from the husk function in Listing 40.

```
1 memblock_alloc_device(ctx, node_id, &x, husk_params.x.
      size);
2 cuMemcpyPeer(x.mem, ctx->cuda.nodes[node_id].cu_ctx,
3                 husk_params.x.mem,
4                 ctx->cuda.nodes[0].cu_ctx,
5                 husk_params.x.size);
```

function, the need for which also becomes apparent in the following sections.

Though the parameter structure has been brought into the scope of the generated function we must also ensure that the actual parameters are brought into scope under their original name. For example for the husk function in Listing 40 the C code shown in Listing 43 will be early in the generated function. Notice that the declaration of the variable `x` on line 3 is not assigned to the corresponding item in the parameter structure. This is due to `x` being part of the list of memory blocks to move to the device in full, thus any reference to `x` inside the generated function must refer to a copy of `x` that is on the corresponding device. However, since the item inside the parameter structure must refer to a memory block on the main device we must ensure that its content is transferred to the corresponding device. This is done for `x` as shown in Listing 44 where line 1 shows an allocation of a memory block on the device corresponding to the node identifier with a size equal to that of `x` on the main device, mutating the locally declared variable `x` to the allocated memory block reference. We then copy the content of the husk parameter `x` to the newly allocated memory block by using the CUDA device-to-device memory copy operation `cuMemcpyPeer` where the CUDA device contexts of the source and destination must explicitly be specified. Notice that not all the parameters that reference memory blocks on the main device must be copied to the device corresponding to the node identifier as is illustrated by `src` and `res` not being copied to the device in a whole. These are special as `src` is to be partitioned to prevent a full copy thus doing a full copy defeats the purpose of it, whereas `res` represent the memory block on the main device that will contain the results of all husk to be combined. Likewise, neither of these can occur in `body_code` where for the case of `src` this is ensured by the fully-copied partitions simplification rule described in Section 4.2.1, whereas

Listing 45 Initialization of the auxiliary partition variables `po` and `pm` inside the function generated from the husk function in Listing 40.

```
1 int32_t max_pm = 1 + sdiv32(C(sm) - 1,
2                             ctx->cuda.cfg.num_nodes);
3 int32_t po = node_id * max_pm;
4 int32_t pm = smin32(max_pm, C(sm) - po);
```

`res` is the result of the husk and should therefore not occur inside its body.

With the required part of the caller's scope reestablished inside the body of the generated function we can declare and initialize the auxiliary variables, specifically the partition element-based offset `po` and the number of elements in the partition `pm`. By design choice the CUDA C backend uses a even-partitioning strategy, i.e. the source array is to be split equally between the devices. However, since the number of elements in the source memory block, specified through the expression in `sm`, may not be a multiple of the number of devices this must be taken into consideration when specifying the auxiliary variables. Therefore we let o be the element-wise offset of the partition in the source, let m be the number of elements in the source, let m' be the number of elements in the partition, let i be the node identifier of the corresponding device, and let n be the number of devices. For the even-partitioning strategy we have that

$$o = i\lceil\frac{m}{n}\rceil$$

and

$$m' = min(\lceil\frac{m}{n}\rceil, m - o)$$

where we note that $\lceil\frac{m}{n}\rceil$ is the maximum number of elements of in any partition. With this definition we also note that all but the last partitioning of the source memory block will have the maximum number of partition elements, whereas the last may have up to $n - 1$ fewer elements. With this we can instantiate the auxiliary variables, e.g. for the husk function in Listing 40 this results in the C code in Listing 45. Here we use another auxiliary variable, namely `max_pm`, representing the maximum number of elements in a partition of the husk, as well as the number of nodes being used as specified in the configuration of the given execution of the program. Notice that since C truncates integer division towards zero we utilize an alternative way of doing rounded-up integer division where `sdiv32` is an integer division that rounds towards $-\infty$ as to ensure correct result when the number of elements in the source array is 0. Notice also that even if there is no partition in the husk function, the offset and size may still be relevant to the body of the husk function or to the concatenation of the results so they must still be defined.

With the variables specifying the partitioning initialized the partitioning can finally commence. This is done in a similar fashion to the full copying of memory

Listing 46 The partitioning of `src` from the main device to `part` on the corresponding device inside the function generated from the husk function in Listing 40.

```
1 memblock_device part;
2 memblock_alloc_device(ctx, node_id, &part, C(pbs));
3 cuMemcpyPeer(part.mem, ctx->cuda.nodes[node_id].cu_ctx,
4                src.mem + C(pbo), ctx->cuda.nodes[0].cu_ctx,
5                C(pbs));
```

Listing 47 The copying of the result of the husk function `hres` to be combined with the results of other husk function executions in `res` inside the function generated from the husk function in Listing 40.

```
1 cuMemcpyPeer(res.mem + C(mrbo),
2                ctx->cuda.nodes[0].cu_ctx,
3                hres.mem,
4                ctx->cuda.nodes[node_id].cu_ctx,
5                C(mrbs));
```

blocks from the main device done earlier, but with a new memory block, namely the partition memory block. Thus, for the exemplary husk function in Listing 40 the partitioning of the source memory block `src` on the main device into a new memory block `part` on the device corresponding to the node identifier `node_id` is shown in Listing 46 where we see the declaration of the partition on line 1, the allocation of it on line 2, and lastly the actual partitioning on lines 3-5. Notice that the byte offset of the partitioning and the byte size of the partition is compiled from the expressions `pbo` and `pbs` respectively which are in turn dependent on the partition variables defined previously.

With the entire needed scope established `body_code` is compiled into the body of the generated function. After this the result to be combined with the results of other calls to the generated function resides in `hres` and must be copied into the combined result `res`. Again, this is done using the CUDA peer-to-peer memory copy operation but whereas the previous operations have been from the main device to the device corresponding to the node identifier of a call to the generated function, the combination of the results is done in the reversed direction. For our running example in Listing 40 this is done as shown in Listing 47 where `mrbo` and `mrbs` are expressions representing the byte offset and byte size for the result of the husk function inside the combined result. Notice that there may not be a result to be combined, e.g. if the result is to be reduced over, in which case the results are copied to the host inside `body_code`.

By the end of the generated function all new memory blocks used for the partitions and for memory blocks copied in full from the main device must be freed, i.e. for Listing 40 this results in the C code shown in Listing 48 .

Listing 48 The freeing of all memory blocks containing data copied from the main device inside the function generated from the husk function in Listing 40.

```
1 memblock_unref_device(ctx, node_id, &x);
2 memblock_unref_device(ctx, node_id, &part);
```

To summarize, consider once again the running example of a husk function given in Listing 40 for which the CUDA C backend will generate the C function shown in Listing 49 where the parameter context structure is defined in Listing 41. That is, at lines 8-11 the parameters are brought into the local scope by their corresponding names with lines 13-18 copying the content from the memory block `x` from the main device to the device that the husk function is to execute with. Then on lines 20-30 the partitioning is done by first initializing the auxiliary partition variables `po` and `pm` with which the partitioning is done by copying from the source memory block `src` to the newly allocated partition memory block `part`. With this the body of the husk function, namely `body_code`, is compiled and the result of the husk function execution `hres` is copied into the combined result on the main device. Lastly all memory blocks allocated explicitly by these steps are freed and the generated function terminates.

5.2.1 Over-partitioning

As equally sized partitions may potentially be significantly different in the time they take to be processed we entertain the idea of over-partitioning as an alternative strategy to the even-partitioning strategy currently deployed by the CUDA C backend.

Consider a program with a husk h. Let the function $T_n(a)$ be the time it takes for h to execute with a as input if given n symmetrical devices, using an over-partitioning strategy, and assuming all partitions exhaust device resources. For an arbitrary array a that exhausts given resources the perfect case would have $T_n(a) = \frac{1}{n}T_1(a)$ as the problem would be evenly divided between all devices. Now let b be an array with valid input for h and let b_1, b_2, b_3, and b_4 be element-wise perfectly even partitions of b where their union comprises all of b. Let $T_1(b) = t$, $T_1(b_2) = \frac{1}{2}t$, and $T_1(b_1) = T_1(b_3) = T_1(b_4) = \frac{1}{6}t$. As illustrated by Figure 5 this is not a perfect case with even-partitioning as

$$T_2(b) = max(T_1(b_1) + T_1(b_2), T_1(b_3) + T_1(b_4)) = \frac{2}{3}t.$$

Over-partitioning splits the array into k partitions where $k > n$ and uses dynamic scheduling to assign them to the devices as they are available. Again, consider the husk h and the array b with its partitions. Let $T'_n(a)$ be the time it takes for h to execute with a if given n symmetrical devices, using an over-partitioning strategy, and assuming all partitions exhaust device resources. Notice that for any array a we have that $T'_1(a) = T_1(a)$ as a single device

Listing 49 The function generated from the husk function in Listing 40.

```
1  int name(void *vctx,
2           int32_t node_id,
3           void *husk_params_p) {
4    futhark_context *ctx = (futhark_context *)vctx;
5    husk_context husk_params =
6      *(husk_context*)husk_params_p;
7
8    memblock_device src = husk_params.src;
9    memblock_device res = husk_params.res;
10   memblock_device x;
11   t y = husk_params.y;
12
13   memblock_alloc_device(ctx, node_id, &x,
14                         husk_params.x.size);
15   cuMemcpyPeer(x.mem, ctx->cuda.nodes[node_id].cu_ctx,
16                husk_params.x.mem,
17                ctx->cuda.nodes[0].cu_ctx,
18                husk_params.x.size);
19
20   int32_t max_pm = 1 + sdiv32(C(sm) - 1,
21                               ctx->cuda.cfg.num_nodes);
22   int32_t po = node_id * max_pm;
23   int32_t pm = smin32(max_pm, C(sm) - po);
24
25   memblock_device part;
26   memblock_alloc_device(ctx, node_id, &part, C(pbs));
27   cuMemcpyPeer(part.mem, ctx->cuda.nodes[node_id].cu_ctx,
28                src.mem + C(pbo),
29                ctx->cuda.nodes[0].cu_ctx,
30                C(pbs));
31
32   C(body_code)
33
34   cuMemcpyPeer(res.mem + C(mrbo),
35                ctx->cuda.nodes[0].cu_ctx,
36                hres.mem, ctx->cuda.nodes[node_id].cu_ctx,
37                C(mrbs));
38
39   memblock_unref_device(ctx, node_id, &x);
40   memblock_unref_device(ctx, node_id, &part);
41
42   return 0;
43 }
```

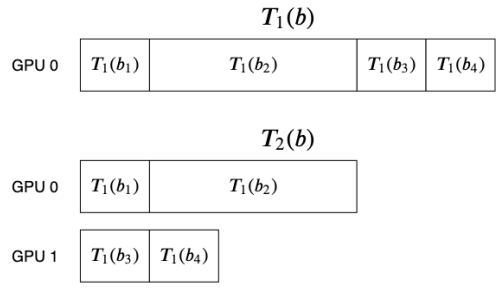

Figure 5: Execution time T_n, where n is the number of devices, of a husk h using the even-partitioning strategy, of partitionsb_1, b_2, b_3, and b_4 of an array b.

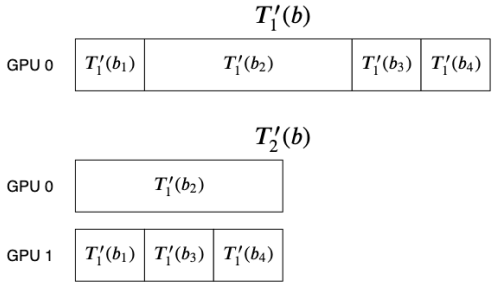

Figure 6: Execution time T_n', where n is the number of devices, of a husk h using the over-partitioning strategy with $m = 4$, of partitionsb_1, b_2, b_3, and b_4 of an array b.

will still have to take the partitions sequentially. As illustrated by Figure 6 this accomplishes the optimal execution time of $T_2'(b) = \frac{1}{2}t$. However, consider an array c with perfectly even partitions c_1, c_2, c_3, and c_4, where $T_1'(c) = t$, $T_1(c_3) = \frac{1}{2}t$, and $T_1(c_1) = T_1(c_2) = T_1(c_4) = \frac{1}{6}t$. With c we again reach an execution time of $T_2'(c) = T_2(c) = \frac{2}{3}t$ where over-partitioning is no better than the even-partitioning as the expensive partition is selected late in the execution of h which is illustrated in Figure 7. Notice that this problem can be somewhat mitigated by increasing k at the risk of the partitions not exhausting device resources leaving precious compute units idle.

Though over-partitioning in theory has equivalent or better execution time than the even-partitioning strategy implementing it would introduce additional overhead. The most obvious added overhead is the increased communication between host and device. For the even-partitioning strategy the host only has to issue all needed operations, such as memory copying and kernel launching, n times whereas over-partitioning must do this k times, where k is likely magnitudes larger than n. The overhead of copying partitions by need could poten-

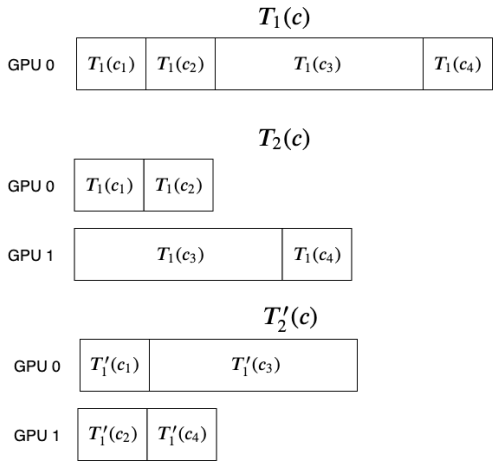

Figure 7: Execution time T'_n and T_n, where n is the number of devices, of a husk h using the even-partitioning strategy and the over-partitioning strategy with $m = 4$, respectively, of partitionsc_1, c_2, c_3, and c_4 of an array c. Notice that for all arrays a we have that $T'_1(a) = T_1(a)$.

tially be mitigated by instead copying the entire memory block to all devices, but this will however increase the amount of memory needed to be transferred for partitions by a factor n and would defeat the primary purpose of the partitions in general. In addition to this whereas for even-partitioning the host can issue device operations and continue executing the host code while the devices executes the commands given, in the case of over-partitioning the host would have to wait for the execution to finish on each partition before continuing with the next, introducing idle time for the device between each partition.

5.3 Worker-thread Runtime Environment

Originally the implementation strategy of husks inside the CUDA C backend used a loop iterating over the number of GPUs to use and running the husk function for each. Since the launching of CUDA kernels does not synchronize the execution of the kernel with the host, i.e. the calling host thread is allowed to continue executing the host program while the GPU executes the kernel, this did in theory allow for parallel execution of kernels. However since the results of the husk function may be copied to the host inside the husk function, which will in turn synchronize the host with the device, the loop-based implementation could introduce sequential execution of the kernels defeating the purpose of using multiple devices. In principle knowing that this is only a problem for the results of the husk function to be reduced the device-to-host copies could be deferred to after all kernels have been launched but this introduces an unwanted restriction on the body of a husk, specifically that it must not contain any CUDA

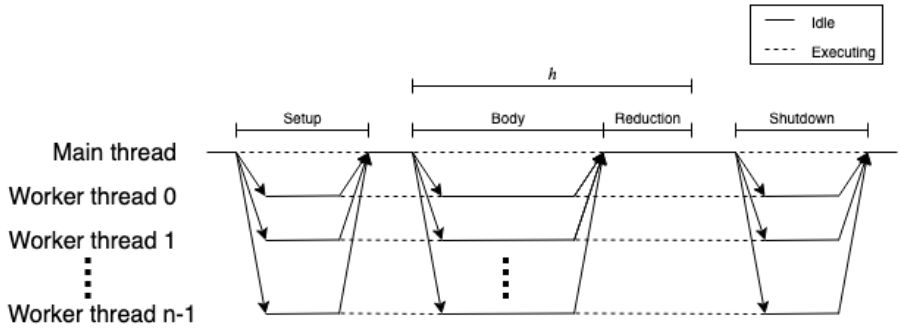

Figure 8: The execution of a Futhark program compiled using the CUDA C backend with the worker-thread implementation using n GPUs.

operations that synchronizes the host and the device which is likely to prove problematic when the capabilities of husks are expanded in the future.

The restriction that the loop-based implementation of multi-GPU husks would introduce leads us to the alternate implementation, namely a worker-thread implementation. The worker-thread implementation is a change to the runtime environment used by the generated program and is based on the realization that the CUDA device context stack is thread-specific, i.e. each executing thread in the host process has its own CUDA device context stack and can therefore issue CUDA commands through different contexts without interfering with each other [1, p. 73].

The worker-thread implementation introduces a number of threads into the generated CUDA C runtime environment equal to the number of nodes. Each of these threads are in charge of setting up their corresponding node context and by association also its CUDA device context. Upon executing a husk the main thread will send a message to each worker thread with the needed information to be able to run the husk function, including any arrays that will contain the results. Upon completing the execution of a husk function the worker threads synchronize and the main thread proceeds by applying the reduction to the results. By the end of the programs execution the worker threads cleans up their corresponding node context and shut down.

Consider a Futhark program that when compiled using the GPU pipeline of the Futhark compiler contains a single husks h being executed in succession. Figure 8 illustrates the execution of this program compiled with the Futhark CUDA C backend using the worker-thead implementation and n GPUs. Notice that having more husks in the program would introduce similar execution snippet as that of h between the setup and the shutdown of the worker threads. However, between execution of husks the worker threads will run idle as the main node is awoken, executes the reduction, and potentially begins the execution of following husks.

In the Futhark CUDA C backend we will be using the pthreads library in the GNU library collection primarily because the backend already uses the

GNU C compiler to compile the generated C program so the acquisition of the pthreads library should not be difficult on most machines using the backend. The pthreads library supplies an implementation of POSIX threads, commonly known as a pthread, which are implementations of the standardized execution model for parallel C threads as specified in the IEEE POSIX 1003.1c standard [6]. Unlike the fork system call on Linux machines that spawn a new identical process at the point of entry, a pthread starts its execution as a call to a specified function with a single void pointer parameter. Therefore in order to be able to spawn worker threads we must define a function as an entry point for the worker thread and a structure with all the information needed, which are; the node identifier of the worker thread, the Futhark context, and the compiled PTX program with all kernels to be executed. Notice that with the node identifier, the Futhark context, and through it the CUDA context, each worker thread is able to access its corresponding node context.

With this structure we spawn each worker thread through their own instantiation of the main worker thread function with the relevant info. The worker thread must then first setup the CUDA device context, load the CUDA module, and initialize the free-list. In addition to these the worker thread must setup a semaphore that the main node will use in order to send messages to the worker thread after setup which we will refer to as the *message signal*. Then, like previously, all kernels are gathered from the module and all static arrays are allocated and initialized on each worker thread. After this has been done all the worker nodes synchronize with the main thread through what we will denote as the *synchronization point*. The synchronization point is a pthread barrier that blocks all waiting threads until the specified number of threads are actively waiting which for the synchronization point is the number of worker threads and the main thread.

When synchronicity has been reached the worker threads will wait for the message signal to be set whereas the main node sets the top of its CUDA driver context stack to the context corresponding to that of worker thread 0 in order to allow for kernels to be launched outside of husks through the main thread. Then the main node enables any interconnects between the CUDA device context 0 and all other worker thread CUDA device contexts as to allow for the fastest peer-to-peer memory copies possible. Notice that any inter-GPU memory copy operation in husks are done either to or from the device of worker thread 0, i.e. the same that is used by the main thread. With this done the main thread starts the execution of the program as usual.

With the worker threads up and running a number of messages are defined for when they are awoken by their respective message signals, namely:

- A synchronization message calling for an explicit device-to-host synchronization.
- A husk message sent when the worker thread must execute a husk function.
- An exit message to be sent when the program is done and the nodes must clean up after themselves.

Listing 50 The definition of a common type for the functions generated from husk functions.

```
1 typedef int32_t (*husk_function_t)(void*,int32_t,void*);
```

Whenever a worker thread is awoken by the message signal it checks the message identifier and executes accordingly.

The first of the messages, namely the synchronization message, instructs all worker threads to do an explicit host-device synchronization before returning to the main thread. Thus when the main thread continues execution all device contexts have finished all queued operations. This is simply implemented by having each worker thread call the CUDA driver API function `cuCtxSynchronize` before synchronizing with the main thread followed by it once again awaiting new messages.

When the exit message is sent the worker threads begin their cleanup phase which involves clearing out the respective free-list, unloading the CUDA module, destroying the CUDA device context, and destroying the message signal semaphore. When this is done the worker thread interrupts its own message loop and dies. After this point any messages sent from the main thread to the worker threads has undefined behavior.

Unlike the previous two messages the husk message is more dynamic and therefore also needs additional information to execute which we will refer to as the content of the message. For the case of a husk message, the message content contains the parameters of the corresponding husk function as specified directly by the husk function, as well as a function pointer to the function generated from the corresponding husk function. This is the primary reason as to why the type signature of the functions generated from husk functions must be generalized as it allows the definition of a common type of this function, namely `husk_function_t` as shown in Listing 50 where the first parameter is the Futhark context structure, the second parameter is the node identifier for the execution of the husk function, and the last parameter is the husk parameter structure corresponding to the husk function. The last parameter is a void pointer as different husk functions have different parameter structures thus representing them as a void pointer allows the function to convert it appropriately. The reason why the Futhark context is passed by a void pointer is due to this type definition being inserted earlier than the definition of the Futhark context structure and therefore the type definition cannot reference it. When a husk message is received the worker thread calls the specified function with the Futhark context, the node identifier of the calling node, and the specified parameters. Upon completion of the corresponding call the result of the call is written to a result variable in the Futhark node context which indicates whether or not the husk body finished successfully. If the result is not 0, which would otherwise signify a successful execution of the corresponding husk, the main thread cleans up and signals the error. Notice that in order to not free on top

Listing 51 An example of an ImpCode representation of a husk.

```
1  imp_husk
2    n_nodes
3    (imp_husk_function
4      husk_f
5      (src, part, pbo, pbs)
6      Nil []
7      [(src, memblock_device),
8       (res, memblock_device),
9       (y, t)]
10     po pm sm node_id
11     {body_code})
12   {rres}
13   {red}
```

Listing 52 The initialization of the auxiliary variable for the number of GPUs to use in the execution of the husk in Listing 51.

```
1  int32_t n_nodes = ctx->cuda.cfg.num_nodes;
```

of each other and to keep all husks in line, the main thread is responsible for freeing all memory from outside the corresponding husk. Since all husks that has safely failed will have freed their own memory before returning a failure and all successful husks will have done the same before ending their execution we can be sure that all memory blocks are correctly freed.

5.4 Launching Husks

With the framework established for the execution of a husk function on multiple GPUs we can finally consider the compilation of an husk in the ImpCode. To make examples of the generated code we consider the ImpCode representation of a husk shown in Listing 51 with a husk function inline.

Inside the backend the occurrence of an ImpCode husk signifies the launching of a husk. However before generating the ground work needed to launch a husk the husk function is used to generate a function representing it, as specified in Section 5.2.

First the auxiliary variable specifying the number of GPUs to use is initialized which is done by assigning it the value of the `num_nodes` in the configuration of the executing Futhark program. As an example of the initialization of the variable representing the number of nodes, consider the ImpCode husk shown in Listing 51 which sets the number of GPUs to use, i.e. variable `n_nodes`, to the value of the configuration variable as shown in Listing 52. Notice that this variable does not generally need to be passed to the function generated from the

Listing 53 The initialization of the parameters used to call the function generated from the husk function of the husk in Listing 51.

```
1 husk_context husk_params;
2 husk_params.src = src;
3 husk_params.res = res;
4 husk_params.y = y;
```

husk function even though the generation of the function uses it to determine partition sizes unless an expression in the husk function uses it. This is due to the variable being accessible through the Futhark context for any generated code that explicitly need to know the number of GPUs to use whereas the auxiliary variable acts as a mediator for the body of the husk function.

Then the C code for the allocation of the host memory blocks for the intermediate results to be reduced is inserted. Since this is done through the ImpCode representation inside an ImpCode husk this is simply done by compiling the attribute.

With this all necessary variables are initialized and the corresponding husk function can be called. However to do such a call the parameters must be inserted into the husk parameter structure corresponding to the generated function for the husk function. Consider again the ImpCode husk in Listing 51. This will create an instance of the parameter structure of the husk function and sets items of it as shown in Listing 53.

Using the message system established in the worker-thread runtime environment we can now call the function generated from the husk function through each worker node by sending a husk message with a pointer to the generated function and the created instance of the parameter structure. When control is returned to the host code, i.e. when the worker threads are done executing the specified function, the reduction `red` is compiled inline to be done by the host.

To summarize, for the ImpCode husk in our running example shown in Listing 51 the execution of a husk is done as shown in Listing 54. In line 1 we see the initialization of the variable representing the number of GPUs whereas the allocation of the memory blocks to contain the results of the husk function to be reduced through `red` is on line 3. Lines 4-7 have the initialization of the husk function parameter structure which is then passed, together with a pointer to the function generated from the husk function, to the worker nodes through the worker-thread runtime environment interface on line 10. Lastly with the worker threads done with the execution of the husk functions the results are then reduced through the reduction resulting from compiling `red` on line 12. Notice that the generation of its husk function is described in Section 5.2 and has therefore been omitted from this example.

As an interesting implementation detail consider the case where an assertion fails inside the function generated from a husk function. In that case, all memory blocks allocated must be freed which is exactly what is done when the assertion

Listing 54 The C code generated by the CUDA C backend for the ImpCode husk in Listing 51.

```
1  int32_t n_nodes = ctx->cuda.cfg.num_nodes;
2
3  C(rres)
4
5  husk_context husk_params;
6  husk_params.src = src;
7  husk_params.res = res;
8  husk_params.y = y;
9
10 send_node_husk(ctx, &husk_f, &husk_params);
11
12 C(red)
```

fails on the main thread. However with multiple threads in flight we cannot have each thread free all memory as they would potentially attempt to free memory that does not exist on their corresponding device or even attempt to do the same freeing of memory simultaneously. Instead when compiling the body of a husk function a failing assertion only frees the memory that has been allocated inside the generated function. By checking the results of the launched husk message, the launching code can then determine if any worker thread failed the call can then free the remaining memory blocks accordingly. Notice that no matter if a failure happens or not, the husk functions will have freed all new memory blocks thus having the main thread free the rest upon failure is enough to ensure all memory is freed.

6 Matching Single-GPU Performance

It is important to realize that the husk operator is agnostic on the number of devices it uses, allowing the number of devices to use to be dynamically determined at runtime. However this introduces the problem of potentially introducing an overhead in the single-GPU case relative to the original backends. Even if the husks introduce decent multi-GPU performance it may not be worth it if the single-GPU case suffers for it so in this section we will discuss some of the overhead introduced by the implementation of husks in the CUDA backend, described in Section 5, and then try to introduce mitigation strategies to minimize or remove it. In Section 7 we measure the remaining overhead.

6.1 Peer-to-peer Memory Copying

In the CUDA C backend all memory blocks used inside the husk functions are copied from the main device to each device to be used for the computation of the husk. Though this is needed in order to actually be able to access the needed data to in turn be able to do their share of the computation, the generalization of this means that the main device also has its partition of the memory copied into its own memory space even though the source array is already there. Notice that though an inter-GPU copy operation is also used to do this copy the API is smart enough to detect that both the specified CUDA device contexts refer to the same device and therefore simply does a device-internal memory copy. Though this is generally faster than a peer-to-peer copy it is still potentially expensive for no gained benefit. To prevent this we can in the case of fully copied memory blocks simply reuse the memory block by having the new memory block reference it directly which is easily implemented by checking if the device in use is the main device.

The copying of partitions can be done in the same way due to the observation that the main node has been given device identifier 0. This not only ensures that the main device can be easily identified through a static check of the identifier but also means that due to the partitioning strategy selected for the CUDA backend, as specified in Section 5.2, the offset of its partition is 0 which allows the partition memory block to be the same as the source memory block. Notice that though an offset larger than 0 would not in theory introduce a problem as the memory pointer used by the partition could just point at the corresponding offset into the source memory block, this would require changes to the internal representation of memory blocks in the generated C code in order to allow offsets into other memory blocks instead of the current reference-counting approach.

Similar to partitions, when the intermediate results of the husk are to be concatenated the device 0 case could also just write the result directly to the memory block of the final concatenated result instead of writing to a newly allocated memory block and doing an internal device memory copy operation. With respect to the implementation this is slightly more difficult than for partitions as the allocation happens inside the body of the husk. Though it should be a general optimization for now, changing the allocation should be kept as an

optimization in the CUDA C backend as we do not want to make assumptions about main device optimizations for future backend implementations of husks.

To make the main device optimization for concatenated results we traverse the body of the husk function backwards and if the allocation is found it is replaced by an if-statement that preserves the allocation when not on the main device, exchanging it for an assignment to the concatenated memory block if not. In some cases the intermediate results to be concatenated are not allocated explicitly but are assigned to another memory block. In that case we continue the traversal but looking for the allocation of the memory block that the intermediate result was set to, replacing it with an assignment to the final concatenated result memory block or looking for its original memory block if it is in turn just an alias for another memory block. There are however some rare cases where this optimization may become more complex, such as if the allocation happens in a conditional body, i.e. in a branch of an if-statement or inside the body of a for-loop, in which case the results may be less predictable as previous allocations for the memory block may not have any relation to the memory block in use at the time of a kernel inside a husk. Notice that the current implementation is naive and simply drops any attempt at changing the allocation whenever it is encountered inside a conditional. Improvements to this optimization is left for future work.

The ability to set the partition and otherwise copied memory to their source in the case of device 0 is not limited to the single-GPU case and can potentially offer increased performance when using multiple GPUs. However since the worker threads are synchronized by the end of the husk operation the performance of the entire husk is limited by the slowest of the worker threads, so having a general performance increase on one GPU may not benefit the performance of the entire husk operation. Notice also that the thread using device 0 was already predisposed to have better performance than the other threads as the now removed peer-to-peer memory copies were previously changed to less expensive device-internal memory copies.

6.2 Inter-thread Communication

Through the introduction of additional threads in the runtime environment generated by the CUDA C backend we also introduce a communications overhead with the main thread sending messages to the worker threads which will in turn wake them up. Though this overhead is generally small compared to the runtime of the husk function it is still an additional overhead relative to the originally generated code that can in fact be mitigated.

In order to accomplish a reduced overhead of the inter-thread communication we first make the realization that the main thread enters a waiting state just after dispatching a message to all worker threads, only continuing upon synchronization with the worker threads when they have finished the corresponding job. Therefore if we relax the notion of worker threads to have the main thread act as a worker thread upon sending a message we gain the benefit of lower communication overhead for that execution. This can be accomplished by generalizing

the handling of messages and decreasing the number of threads by one. With this the main thread then sends messages to all the worker threads, starting with identifier 1, thus reserving identifier 0 for itself. When the messages are dispatched the main thread continues on to handle the message itself for device 0 and synchronizes with the other worker threads once done. This means that when there is only a single device to use the main thread will not spawn nor outsource work to worker threads and will neither need to synchronize upon the completion of handling a message.

Notice that like with the optimization of memory reference setting on device 0 as discussed earlier this optimization does also have performance increase on one of the threads. However this is again on the device with identifier 0 and we again have that the performance gain in the multi-GPU case may be limited by the other threads.

6.3 Forced Direct Index Function

In Section 4.3 we discussed the problem of memory layout through index functions when doing partitions, specifically that the memory copying for partitions may become increasingly complicated with complex index functions for the source arrays. We then introduced two slightly aggressive strategies for ensuring direct index function on the source array, heavily simplifying the memory copying of partitions, but at the cost of either having to restructure the memory of the source arrays or internalizing the partitioning to the body of the husk, increasing the amount of memory to copy. Of these two strategies the second was selected which may have appeared unreasonable at the time but was strongly motivated by the prevention of the overhead on the single-GPU case that the first of the strategies would introduce.

Consider a column-major ordered array a which is the source array of the partition in a husk h. When using the first strategy introduced in Section 4.3 a transposition of a would be done, resulting in a new memory block with an array a' which would in turn have a direct index function. The array a' would then be the new source array of h. Since the partitioning of a source array in the single-GPU case is generally a full copy of the array, no matter the index function, the transposition of a introduces an unnecessary overhead to the single-GPU case. Unlike the previous single-GPU optimizations this cannot be mitigated by a special case for device 0 setting the memory of a' to reference to the memory of a as all access to the memory of a' assume row-major ordering of the array given that was the index function of a'. Likewise it is not possible to make a special case in the explicit allocations step of the compiler to give a' a conditional index function, i.e. having its memory layout remain column-major in the single GPU case and have it row-major otherwise. Technically, in the explicit allocation step when the strategy is employed the husk could be split into two branches of an if-statement; one branch for the multi-GPU case with a' being introduced inside it and having it as the source array of the contained husk, and the other branch for the single-GPU case which would simply be a husk that retains a as the source of the partition. Notice however that this approach would introduce

redundant work for the compiler as well as clutter the generated intermediate representation of the encapsulating program. Notice also that this would require the number of GPUs to be defined outside of husk operations in order to allow the branching.

In contrast by employing the second strategy introduced in 4.3 we have that the source array is fully copied if it does not have a direct index function. In the single GPU case this introduces no additional overhead as the partitions would already have been full copies which in turn becomes a reference assignment instead through the optimizations introduced in Section 6.1.

6.4 Reduction on the Host

As mentioned in Section 3.2 the reduction of a husks intermediate results is done on the host. This means that the results to be reduced must be transferred to the host upon completion of the individual husk function executions. Previously the results of reduction operations were allowed to stay on the GPU if it was not used on the host but with the host-level reduction introduced on these reduction operations by the husk transformation of the reduce SOAC introduced in Section 3.1.2 it will be copied to the host no matter the future use of it. This is generally the effect of the design choice made of the host-level reduction and since reduction results are often scalars this transfer itself is mostly inexpensive.

However to understand why this deviation can become expensive, consider a program that computes a reduction r the result of which is needed on the GPU in an adjacent operation o but will never be used on the host. Previously r would be a standalone operation and the execution could be summarized as:

1. The host launches the kernel that computes r and let the result stay on the device, potentially by dispatching a memory copy internal to the memory of the GPU.

2. The host launches the kernel that computes o with the result of r that is already on the GPU.

However with the husk operator r will be in the body of a husk, so the execution is:

1. The host distributes r over the GPUs to use.

2. The results of each distribution of r is copied into a buffer on the host.

3. The buffer with the results are reduced using the reduction monoid of the husk.

4. The result of the husk reduction is copied to the main GPU.

5. The host launches the kernel that computes o with the result of the husk that was recently copied to the main GPU.

For the GPUs the copy to and from the host is negligible assuming the result is a small array or a scalar. However, on the host the first of these executions would not require any host-device synchronization between the computation of r and the computation of o so the host is free to launch the kernel for o right after launching r, but for the the execution where r is inside a husk the host copies the data from the GPUs which introduces a host-device synchronization as the host has to both wait for r to finish and for the results of r to be copied to its memory. In the case where multiple GPUs are used to compute r this is necessary as the reduction changes the final results. However, for the case where r is computed on a single GPU and therefore is not distributed we have that the husk reduction leaves the intermediate result unchanged due to the definition of monoids. Therefore in the single-GPU case the reduction on the host becomes unnecessary and the result could be allowed to stay on the host.

The optimization for the single-GPU case for results of reductions that can remain on the main GPU is more difficult than the other optimization however as it effects the usage of the result outside the husk operator which would in turn be dependent on the number of GPUs in use. This could in principle be accomplished in a similar way to how reduction results are allowed to stay on the GPU that is by letting the memory space of the result be the memory space of the main device. This would mean that for the single GPU case we could simply let the result stay on the main GPU but for multiple GPUs the intermediate results would be copied to the host, reduced, and the final result being transferred back to the device. The primary problem with this optimization would be that, if the result is only used on the host the result would be transferred back to the host which would be redundant work in the multi-GPU case. Since this should optimize the single-GPU case it should be considered for future work.

Part III
Evaluation and Final Remarks

7 Performance Comparison and Benchmarking

The point of the husk operator is to be able to efficiently distribute a problem in a Futhark program between multiple GPUs, taking advantage of the additional resource that the additional devices offer. In this section we will evaluate the performance of a selection of Futhark programs, compiled using the CUDA C backend, using multiple GPUs, and comparing it to the performance of the same programs using a single GPU. However before doing this comparison we first compare the single-GPU performance of Futhark program compiled using the CUDA C backend with the husk operator implemented to the currently released Futhark compiler CUDA C backend, investigating select performance differences introduced by the husk operator implementation.

Throughout this section we will be using a selection of the established Futhark benchmark programs[1]. All benchmarking is conducted on a machine with the following specifications:

OS Microsoft Windows 10 Pro - Version 10.0.18362 Build 18362.

CPU AMD Ryzen 2700X with 8 cores and 16 threads each running at a base clock of 3.7 GHz and a max turbo clock of 4.3 GHz.

RAM 32 GB of DDR4 RAM running at 3200MHz.

GPU Two NVIDIA GeForce RTX 2080 each with 8 GB memory, a memory bandwidth of 448 GB/s, and 2944 CUDA cores running at a max core clock of 1710 MHz. The two discrete GPUs are connected to the PCIe bus through 8 lanes each and are interconnected using an NVLink bridge with at 50 GB/s bidirectional bandwidth making it 25 GB/s in each direction [20].

7.1 Comparing Single-GPU Performance

To assess the effect that the addition of the husk operator has on the single-GPU execution of compiled Futhark programs we want to take a general look at the performance difference between the compiler with and without the husk operator. To accomplish this we use the Futhark benchmarking utility to run the benchmarks in the Futhark benchmark collection, excluding the micro-benchmarks as they currently fail on Windows. We do this for both the currently released version of the Futhark compiler and the new Futhark compiler that generates husks, both using the CUDA C backend. Appendices B.1 and B.2 show the results of these benchmarks, displaying the average runtime of 100 runs for each of the data sets of each of the benchmark programs. For each benchmark program let r_{n1} and r_{orig} be the average runtime over the respective data sets of the program compiled with and without husks respectively, with which we define the average slowdown as r_{orig}/r_{n1} and the average speedup as r_{n1}/r_{orig}. To

[1]The Futhark benchmark programs can be found at https://github.com/diku-dk/futhark-benchmarks.

summarize the performance difference of the benchmark programs, the average slowdown and average speedup greater than 1 is shown in Figure 9. With the addition of the husk operator we would expect that the comparatively restrictive structure of the program it introduces would be cause for overhead in some programs but we would not expect increased performance. However though Figure 9 shows that the difference in performance of most of the programs is negligible we actually see cases for both slowdown and speedup with programs such as NN and LUD in the Rodinia benchmark set being respectively $\sim 2.5\times$ and $\sim 1.9\times$ slower than without the husk operator and programs such as CFD also in the Rodinia benchmark set gaining a $\sim 1.7\times$ performance increase. To gain a better understanding of this we investigate the most glaring examples of both speedup and slowdown.

7.1.1 Rodinia NN

Of the performance decreases shown in Figure 9 the largest is that of the NN Futhark implementation based on the implementation of the k-nearest neighbor data analysis algorithm from the Rodinia application suite [2]. This benchmark program becomes $\sim 2.5\times$ slower with the addition of the husk operator, which is a product of the reduction of the husk being done on the host.

Consider Appendices C.1.1 and C.2.4 where we see a summary of the CUDA API calls and GPU activities done while running the NN benchmark program with the supplied medium.in data both with and without the husk operator being used by the compiler. We see that the number of calls to the kernels as well as their execution time stay the same between the executions but where the program compiled without husk operators have 200 memory copies inside GPU memory, the husk operator replaces these with 100 device-to-host and 200 host-to-device memory copies. This is in fact the product of the overhead introduced by the husk reduction done on the host that remains unmitigated in the single-GPU case as discussed in Section 6.4. As mentioned the actual difference in time spent on the GPU between the internal GPU memory copy and the device-to-host-to-device copy is negligible but the cost comes from the the unnecessary synchronization done on the device-to-host synchronization that halts the execution of the active CPU thread until the reduction kernel finishes and the data has been transferred to the host. Notice that the reason for the number of device-to-host copies only increasing by 100 is due to one of the two results of the reduction being copied to the host in the version without husks as well but since it also exists on the GPU it is not transferred back for the next execution of the reduction.

7.1.2 Rodinia CFD

In the opposite side of the scale we have the CFD benchmark program, also based on one of the applications in the Rodinia suite, that actually gains a $\sim 1.7\times$ speedup in performance with the husk operator. Again we turn to the NVIDIA profiling tool to gain a better understanding of difference the husk

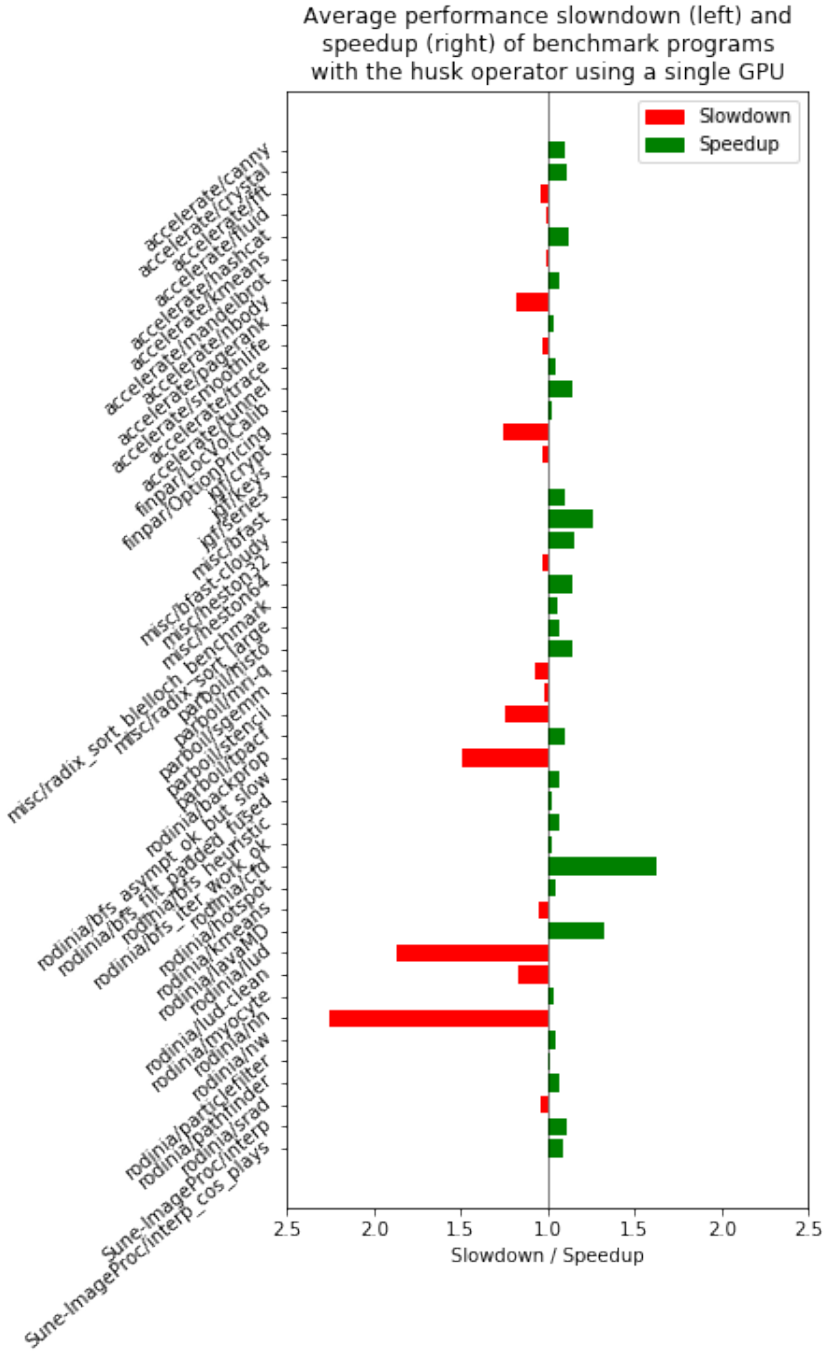

Figure 9: Average slowdown and speedup magnitude of the benchmark results in Appendix B.2 relative to the results in Appendix B.1.

operator makes to the generated program. Appendix C.1.2 and C.2.5 show a summary of profiling the CFD benchmark program using the data set fvcorr.domn.193K.toa.gz supplied with the benchmarks. From the profiling results we note three interesting differences:

1. The program with husks have 12000 fewer transposes as the program without husks.

2. The program with husks launches a large number of copy kernels which the other program has none of.

3. The segmap kernels of the program with husks are faster than the corresponding kernels in the program without husks.

The additional copy kernels are caused by the internalization of partitions, either through the second simplification rule described in Section 4.2.1 or by the partitioning strategy described in Section 4.3. Since this changes the index function of some of the arrays in the program, i.e. with a different offset and a size, the compiler determines that select transposes of 3-dimensional arrays, corresponding to the 12000 missing transposes comprising 6000 transpose compositions, can be written as copy kernels which are in turn faster than the transpose compositions.

In addition to combining transpose compositions into combined copy kernels, the compiler changes the layout of the resulting memory block slightly. The result of this slight restructuring becomes very apparent by the third observation made from the profiling results as it changes the access pattern made to these memory blocks to be coalesced, improving the performance of some of the memory accesses in the significantly faster segmap kernels.

Notice that improved performance of the CFD benchmark program is a byproduct of the husk operator as it is primarily caused by a slight index function change of select arrays causing the compiler to select an alternative way of restructuring the arrays. Therefore the performance improvement could potentially be made an optimization in the compiler separate from the use of the husk operator.

7.2 Multi-GPU Performance

Though good single-GPU performance is important, the goal of the husk operator is to allow for the utilization of multiple GPUs. However we do not expect all programs to actually benefit equally from the husk operator and some may take a performance hit due to the expensive inter-GPU memory copies potentially outweighing the decreased execution time of husk body.

Since we do not need nor expect the performance of multi-GPU execution of Futhark programs to be as consistently close to the single-GPU execution we will only consider select benchmark programs, discussing programs that increase and decrease in performance from the husk operator on two GPUs compared to just one.

Data set	Avg. Runtime (ms)		Rel. Perf.
	One GPU	Two GPUs	
small.in	86721.180	88224.219	98.296%
medium.in	66455.344	51178.730	129.849%
large.in	745787.188	399335.906	186.757%

Table 3: Performance comparison between using a single and two GPUs when running the benchmark data sets for the LocVolCalib benchmark program, based on the results shown in Appendix B.2 and B.3.

7.2.1 Finpar LocVolCalib

The first of the benchmark programs we consider is the LocVolCalib program in the collection of parallel financial benchmarks. A performance comparison of LocVolCalib using two GPUs and a single GPU is shown in Table 3 based on the benchmarking results shown in Appendix B.2 and B.3. We see from the relative performance that nothing is gained from using two GPUs for the data set small.in but for medium.in the performance increases by $\sim$ 30%, whereas the performance increases $\sim$ 87% with the large.in dataset. This illustrates a fundamental limitation of the husk operator, specifically that if the distributed problem does not exhaust the resources of a single GPU, using more symmetric GPUs comes with no benefit as is the case for small.in and to some extent medium.in. However notice that relative to a single GPU the maximum performance increase of using more symmetric GPUs is proportional to the number of GPUs used, which the relative performance in Table 3 for the large.in data set gets close to.

To understand why LocVolCalib performs so well with two GPUs, consider the NVIDIA CUDA profiling results for running LocVolCalib with the large.in data set and using one GPU and two GPUs as shown in Appendix C.2.1 and C.3.1 repectively. From this we see how the kernels are almost perfectly distributed between the two GPUs, i.e. the runtime of the kernels takes about half the time to execute on each GPU relative to the corresponding kernels when on a single GPU. As illustrated by the relative performance shown in Table 3 we do not gain a perfect doubling of performance, but given that the inter-GPU memory copy for this program is inexpensive it is likely that performance may increase further with even larger data sets.

7.2.2 Accelerate Tunnel

Another example of a well behaving benchmark program with respect to multiple GPUs is the Tunnel benchmark program in the Accelerate benchmark collection. Table 4 shows the relative performance of the Tunnel benchmark program between executing it with one GPU and two GPUs. The relative performance tells us that by increasing the second and third parameter to the program while keeping the first parameter at 10 the performance increase is allowed to grow up to a $\sim$ 50% increase in performance. However it also shows that additional

Data set	Avg. Runtime (ms)		Rel. Perf.
	One GPU	Two GPUs	
#1 ("`10f32 800 600`")	350.270	342.460	102.280%
#2 ("`10f32 1000 1000`")	592.560	585.670	101.176%
#3 ("`10f32 2000 2000`")	2352.390	1725.940	136.296%
#4 ("`10f32 4000 4000`")	7873.750	5357.360	146.971%
#5 ("`10f32 8000 8000`")	31266.711	20770.920	150.531%
#6 ("`10f32 16000 16000`")	124773.000	83034.828	150.266%
#7 ("`10f32 32000 16000`")	249784.484	166037.625	150.438%
#8 ("`10f32 16000 32000`")	249938.047	166310.328	150.284%
#9 ("`100f32 16000 16000`")	511468.820	277573.240	184.264%

Table 4: Performance comparison between using a single and two GPUs when running the benchmark data sets for the Tunnel benchmark program, based on the results shown in Appendix B.2 and B.3.

performance can be gained by increasing the first parameter.

To understand why the parameters affect the performance increase we consider profiling results of the last two data sets shown in Table 4 as shown in Appendix C.2.2 and C.2.2. We see how, with the second and third parameters being large, the inter-GPU memory copies correspond to $\sim 24\%$ of time spent on the device where the remaining time is spent on the only kernel of the program, which in turn appears to be well distributed as each GPU uses around half the time of the corresponding kernel when using a single GPU. In fact it appears that the amount of memory used by the program is based on the last two parameters and as they increase the time needed for executing the kernel and the time required for the inter-GPU memory copy both increase, explaining the $\sim 50\%$ ceiling when only increasing the last two parameters. However increasing the first parameter only increases the execution time of the kernel, allowing the performance to increase further. This benchmark program illustrate that husks may work better for some data sets than others as a product of husks being primarily limited by data transfers.

7.2.3 Accelerate Mandelbrot

With the Tunnel benchmark program we saw how the performance gain resulting from the increase of the last two parameters of the program was limited whereas additional performance gain was reachable when the first parameter increased. The benchmarking data sets used were fortunate as the limit on the performance was actually above the performance when using a single GPU, but this is not the case for all programs and all data sets. An example of this is the benchmark program Mandelbrot in the Accelerate benchmark collection of which the relative performance between the use of one and two GPUs based on the benchmarking results shown in Appendix B.2 and B.3 is shown in Table 5. Here we again see how an increase in two of the parameters reaches a limit

Data set	Avg. Runtime (ms)		Rel. Perf.
	One GPU	Two GPUs	
#1 (“`800 600 ... 100 16f32`”)	205.920	332.640	61.905%
#2 (“`1000 1000 ... 100 16f32`”)	255.020	356.430	75.548%
#3 (“`2000 2000 ... 100 16f32`”)	753.950	865.870	87.074%
#4 (“`4000 4000 ... 100 16f32`”)	2403.320	2687.040	89.441%
#5 (“`8000 8000 ... 100 16f32`”)	7746.360	9235.570	83.875%
#6 (“`16000 16000 ... 100 16f32`”)	30183.510	35808.031	84.293%
#7 (“`16000 16000 ... 1000 16f32`”)	144998.562	92670.070	157.468%
#8 (“`16000 16000 ... 10000 16f32`”)	1278715.500	661512.750	193.302%
#9 (“`32000 32000 ... 100 16f32`”)	118524.531	142400.938	83.233%

Table 5: Performance comparison between using a single and two GPUs when running the benchmark data sets for the Mandelbrot benchmark program, based on the results shown in Appendix B.2 and B.3.

but this time the limit is actually around 15% below the program using a single GPU meaning that utilizing more resources actually comes with a cost rather than a benefit. However we also see that increasing the second-to-last parameter allows the performance to increase to almost the theoretical limit of double the performance, further reinforcing the idea that the benefit from multiple GPUs with the husk operator is dependent on the input data in some programs, but also introduces the notion of lost performance for some input data.

Inspecting the profiling results for the last two data sets used in Table 5 as shown in Appendix C.2.3 and C.3.3 we see that for the best case, i.e. the data set with the second-to-last parameter being large, the inter-GPU copy operation takes up only $\sim 3\%$ of the total execution time on the GPU. However we also see that for the data set where the first two parameters are large the inter-GPU memory copy becomes the dominant operation on the main GPU by taking up $\sim 60\%$ of the execution time which together with the kernel amounts to more execution time than for the single GPU case.

7.2.4 Rodinia CFD

As we have seen, the transfer of data between GPUs can limit the performance of husks but whereas the Mandelbrot benchmark program was able to gain performance with the right data, some programs rely on computations with so much data that the overhead of the transfer becomes consistently dominant. For an extreme example of this, consider again the CFD benchmark program which we saw gain increased performance on a single GPU when using the husk operator, as discussed in Section 7.1.2. The relative performance between the use of one and two GPUs for running the CFD benchmark program based on the benchmarking results shown in Appendix B.2 and B.3 is shown in Table 6 which illustrates that the program using two GPUs takes $\sim 5\times$ the time to execute compared to when using only one GPU. As the profiling results in

Data set	Avg. Runtime (ms)		Rel. Perf.
	One GPU	Two GPUs	
fvcorr.domn.097K.toa	1187663.875	5471266.000	21.707%
fvcorr.domn.193K.toa	1774197.750	7837625.000	22.637%

Table 6: Performance comparison between using a single and two GPUs when running the benchmark data sets for the CFD benchmark program, based on the results shown in Appendix B.2 and B.3.

Appendix C.3.4 shows, the culprits of the poor performance are the 36002 inter-GPU memory copy operations taking up $\sim 88\%$ of the execution time on the main GPU. This is primarily caused by the heavy use of memory inside map operations but is made worse by the need to internalize the partitions due to indirect index functions increasing the amount of memory to copy.

8 Related Work

In this section we take a look at select existing work, comparing it to related aspects of the husk operator implementation.

8.1 Automatic Parallelization and Data Distribution

A large body of work has been dedicated to automatic parallelization of code, using approaches ranging in a spectrum from fully dynamic analysis to fully static analysis.

The husk operator accomplished distributed parallelism through the transformation of the map and reduce SOACs in Futhark, which in turn are explicit in their parallelism. This arguably makes the extraction of the husk operator a static analysis of the program which in turn results in a potentially higher level of parallelism exploitable by the program through distribution of the operation.

On the other end of the spectrum we have various dynamic analyses, a large section of which focusing on loop-level parallelization, i.e. attempting to discern independent loop iterations that can in turn be done in parallel. An example of such analysis is the inspector-executor model, in which the inspector determines, at runtime, the groups of iterations in a loop that can be executed in parallel which is then exploited by the executor [30]. Building on this, the inspector-executor model can be used to reorder both the data and iteration space in a way that optimizes communication overhead or locality of reference [8, 34]. Another example of dynamic analysis for the purpose of loop-level parallelization is software thread-level speculation (TLS) which executes the loop iterations in parallel and out of order, not considering any data dependencies between the iterations but fixing any faulty executions of iterations upon detection using strategies resembling an extended cache coherence protocol [32, 29, 35, 18]. To reduce the speculative memory footprint and cache layout of TLS, various optimization can be applied [22] which has in turn been used in a distributed setting in order to optimize communication overhead [21].

Though such fully dynamic analyses are very general and capable of extracting partial loop parallelism, they exhibit significant runtime overhead and may require expensive intercommunication between the nodes in distributed systems, such as when a speculation in TLS fails. To alleviate this overhead, subsequent approaches have been aimed at creating hybrid approaches, combining static and dynamic analyses, typically under the form in which the static aspect of the analysis is responsible for extracting the predicates that the dynamic part of the analysis then verifies cheaply at runtime. In relation to this, a number of studies have been conducted on various techniques for summarizing the set of array indices accessed inside loops, in turn motivated by the fact that point-to-point dependence analysis is impractical for analyzing large loops containing complex control flow, multi-nested inner loops, and function calls. The primary idea for this stems from the fact that array references are aggregated in some abstract-set representation up to the level of the target loop with which the independence between iterations is modeled. For example SUIF [10] supports inter-procedural

analysis and uses a polyhedral representation which is relaxed by extracting sufficient conditions for independence from branch conditions that are in turn tested at runtime [19]. Due to the typically restrictive nature of the polyhedral representations, other work introduces alternative representations such as the integer-set framework proposed by Adve & Mellor-Crummey as implemented in the Rice dHPF compiler [3]. Likewise Rus, Hoeflinger, & Rauchwerger [33] has proposed a language to accurately represent array summaries by introducing symbolic nodes, such as union, intersection, gates, function calls, whenever the set operations would fall outside the baseline abstraction for representing array slices. While loop parallelism could in principle be decided on this representation, e.g. by inspectors that evaluate the set expression, a more efficient and scalable approach has been to translate the set equation to a parallel-predicate language [24] which encodes the sufficient conditions under which the equation holds. For example such techniques can prove loop parallelism by exploiting the monotonicity of indirect-array indices [23] which is unknown at compile-time.

Though the mentioned work offers approaches for automatic parallelization, some can also infer a partitioning of data for the nodes in a distributed setting, minimizing the overhead of the communication between the nodes. This is related directly to the partitioning of data for the husk operator on multi-GPU systems which is inferred statically through the transformations of the map and reduce SOACs as discussed through the SOAC transformations in Section 3.1. Additionally the husk operator uses a pessimistic strategy for used memory blocks that are not partitioned by broadcasting them to all GPUs being used.

An alternative static strategy for identifying the data to transfer between nodes in a system is through *symbolic range analysis* [5, 11] which can be used for determining the range of values a variable can contain at a given point in a program. A symbolic range analysis can prove useful for a plethora of static analysis, such as static branch prediction as described by Patterson [26] and compile-time verification of program properties to prevent runtime errors, such as index bound errors as used in the Whiley programming language [27]. With respect to partitioning specifically, symbolic range analysis can be used to determine the memory regions to potentially be accessed by a program as described by Rugina & Rinard [31], by Yong & Horwitz [36], as well as by Paisante et. al. [25], and has been used for sectioning of data specifically for automatic parallelization on distributed memory spaces as is done in the Rice Parallel Fortran Converter [12] combining it with *regular section analysis* as proposed by Callahan [7]. As the SOACs of Futhark allow for direct identification of some partitions, using symbolic range analysis to determine these would only complicate transformations but could potentially work in tandem with the current transformations to determine not only other unidentified partitions but also limit the amount of data to transfer between devices by restricting it to only the memory regions that the analysis deems to potentially be accessed.

8.2 Futhark

Similar to the introduction of the internal husk operator, Futhark has seen an increasing body of work focusing on optimizing the parallel performance of the language. The most recent work at the time of writing is the addition of a compiler-driven analysis called *incremental flattening* that at runtime determines the optimal parallelism of the different levels of hardware exposed by the system it is executed on [13]. Relating incremental flattening to the husk operator we might argue that the distributed setting that the husk operator introduces is in fact an additional level of hardware, but due to the isolated nature of the operation inside the husk body, the Futhark compiler will instead apply incremental flattening at the same level as previously which would in turn be optimized for each device in the system. Additionally this may benefit incremental flattening in the future as it gives a distinction between the devices used through the isolated execution of husk functions. Notice however that with the current restriction of symmetric devices in the CUDA C backend the decisions made at runtime by the auto-tuner should not be different between the different calls to the husk function but could become with more relaxed device selection rules.

Another example of optimizations done in the Futhark compiler is the redomap construct [16] which has been shortly discussed in Section 4.1. The redomap construct is an internal operator, similar to the husk operator, which represents map-reduce compositions produced through fusion of select patterns in programs. The redomap construct is used to optimize the reduction over an array produced by a map operation by combining the kernels and storing intermediate results in fast memory.

9 Conclusion

We have presented the internal husk operator to the Futhark compiler, allowing the intermediate representation of a Futhark program to express distributable operations and data. This has in turn allowed the extension of the existing CUDA C backend of the Futhark compiler, enabling the use of multiple GPUs through isolated execution of the body of the husk in a newly introduced worker-thread runtime environment, combining the results either through distributed concatenation or through reduction after synchronization. In order to minimize the overhead introduced to the single-GPU case for husks, we have discussed and implemented optimizations bringing the generated program closer to the programs without the husk operator when only using a single GPU.

Through the use of the established Futhark benchmark suite, we have seen examples of close to perfectly linear performance scaling of select programs using large data sets, seeing up to 1.93× performance when using two GPUs compared to when using only one. Likewise, we have shown programs where the performance only increases little and in some cases decreases when introducing more GPUs, primarily due to a dependence on large amounts of data being transferred between the GPUs. Using the same benchmark programs we have also compared the performance of the single-GPU case with and without the husk operator, which showed that the performance generally stays the same except for a few outliers displaying both significant speedups of up to $\sim 1.7\times$ and severe slowdowns of up to 2.5×. Though the slowdowns may make the current state of husks a hard sell, we have discussed mitigation strategies that may in turn bridge the gap further.

9.1 Future Work

Though this implementation of the husk operator arguably shows potential, it is still in its infancy and there are still many optimization and use cases left unexplored. In this section we introduce a select subset of potential future work.

9.1.1 Generality of Husks

Though we have only explored the use of husk operators for multi-GPU systems in this thesis, husks are not specifically catered towards only that purpose. In fact, the actually purpose of the husks are determined by the Futhark backend receiving them as long as the semantics are retained.

An example of an alternative use for the husk operator is distributed computing, i.e. not a single machine but a network of machines with each their own hardware. This is not far from the notion of multi-GPU systems as they also act as co-processors with each their own memory space. However, in the setting of distributed computing the focus is not always upon performance, but may also be restricted by the space available on each machine, especially given the relatively narrow bandwidth on the interconnect between machines. Since the

machines in such a network may have varying resources, such an implementation may require a different partitioning strategy than just splitting equally between machines and may additionally benefit from a decentralized execution model rather than the centralized execution model currently employed by husks, as to not require a main node to have enough storage space for all the data relevant for the execution of a program.

9.1.2 Worker-thread Implementation Adaptation

As a byproduct of the introduction of the husk operator, a worker-thread runtime environment has been added to the C code generated when using the CUDA C backend of the Futhark compiler. Though the messages in this runtime environment has been primarily catered towards the use of husks, the implementation could in principle be easily altered for alternate multi-thread use cases. An example of such alteration would be the addition of a mutli-threaded C backend to complement the sequential C backend, allowing for utilization of the parallelism exposed by the GPU pipeline of Futhark without having to use GPUs.

9.1.3 Further Optimization of the Single-GPU Case

As discussed in Section 6 there are still optimizations left to be done for the single-GPU case. One such optimization is allowing the reduction result to remain on the device when only a single GPU is used, as described in detail in Section 6.4. Another optimization builds upon the current optimization described in Section 6.1 that attempts to use the final concatenated result memory block for the result of the intermediate result to be concatenated from the main GPU. The current version of this optimization tries to track the memory block for the intermediate result back to the allocation of it, but fails if any assignments or allocations in the chain are in conditionals. This naive approach may be improvable through more analysis of the conditionals or through an alternative approach.

9.1.4 Inter-husk Data Reusage

As illustrated by the benchmarking results in Section 7.2, the primary culprit of poor performance when using multiple GPUs is the memory being transferred between GPUs. Though this is necessary to be able to execute kernels correctly, there can be cases where the data has already been copy to the GPU for the execution of a previous husk. Therefore, partitions and broadcasted arrays that we can be sure have not been altered since the last time it was copied can potentially be reused. Likewise, the results of husks to be concatenated may in turn be used as partitions by other husks, giving way for yet more reusable data already on the GPUs.

One way of determining what data can be reused is to do it dynamically at runtime. An example of an approach, that was in fact shortly attempted,

Listing 55 An example of a Futhark program with a `map` operation where only sections of the arrays `xs` and `ys` are used inside the lambda function.

```
1 let main [m] (xs: [m]i32) (ys: [m]i32)
2              (inds: [m]i32): [m]i32 =
3   let j = inds[0] % 32
4   in map (\i -> xs[j] + ys[i]) inds
```

is the use of a memory block cache in the backend that each GPU places any broadcasted arrays, partitions, and results to be concatenated into. Whenever a husk is the going to broadcast or partition memory, the cache is checked and if the data is in it the husk function executions continue with that data instead of copying from the main GPU. However, as memory blocks may be written to on the main GPU the cache lines for each device must be invalidated whenever this happens, as their content may become inconsistent with the actual data. In general such caching needs little knowledge about the program at hand and can thus be generalized, but it also comes with flaws, specifically that it adds an overhead at runtime for lookup, pushing, and invalidation of the cache lines.

An alternative way of determining the reusable memory blocks is through static compile-time analysis. As an example this could be accomplished by use of a data-flow analysis, e.g. by use of flow graph algorithm similar to the one described by Allen and Cocke [4]. Such analysis should not introduce any overhead at runtime but there may still be cases where the compiler must assume that data cannot be reused, such as when a write to an array appears inside a conditional even if the write does not happen at runtime. Such analysis could be coupled with some runtime analysis, making decisions at runtime that was indeterminable at compile-time. Notice however that compile-time analysis may require changes to the intermediate representation of Futhark, potentially introducing the notion of the multiple device memory spaces that the husk operator kept the rest of the compiler oblivious to.

9.1.5 Minimizing Inter-GPU Memory Copying

We have seen that the presented implementation of the husk operator bases all partitioning on the transformed SOACs and handles all outside memory blocks used inside the body of the husk pessimistically by making a full copy from the main GPU to the other GPUs, which contributes to the primary culprit of bad multi-GPU performance. There may however be cases where the husk body only uses a select section of the arrays that are not partitioned. An example of this is shown in Listing 55 where we have established that a husk operator would be the map over a partition of `inds`, whereas `xs` and `ys` is fully copied to each GPU. We notice that only a single element of `xs` is ever used, no matter the size of it, whereas depending on `inds` the number of relevant elements in `ys` may vary, which leads us back to some of the analyses discussed in Section 8.1.

For example in Listing 55 the variable `j` can only be in the integer range

$[0, 31]$ which can be determined statically through the use of symbolic range analysis, thus it is enough to only copy the 32 first elements of `xs`, but since the values of `inds` are only known at runtime the section of `ys` to copy cannot be limited at compile-time. Alternatively using variants of the hybrid approaches mentioned, the exact elements used from `xs` and `ys` should be determinable at runtime by determining the value of `j` through the value of the first element of `inds`, as well as all the values of `i` for each partition of `inds`, before execution of the husk functions and thus allowing selective transfer of the corresponding elements. Notice however that the examples are simple and there are more useful, but also more complex, cases, e.g. if the indices to determine the data to be used may be dependent on an operation inside the body of the husk.

References

[1] *CUDA DRIVER API - API Reference Manual*. https://docs.nvidia.com/cuda/pdf/CUDA_Driver_API.pdf, trm-06703-003 _vrelease version edition, July.

[2] Rodinia: Accelerating Compute-Intensive Applications with Accelerators. http://rodinia.cs.virginia.edu.

[3] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. *ACM SIGPLAN Notices*, 33(5):186–198, may 1998.

[4] F. E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3):137, March 1976.

[5] Hansang Bae and Rudolf Eigenmann. Interprocedural Symbolic Range Propagation for Optimizing Compilers. In *Languages and Compilers for Parallel Computing*, pages 413–424. Springer Berlin Heidelberg, 2006. https://engineering.purdue.edu/paramnt/publications/LCPC05.pdf.

[6] Blaise Barney. POSIX Threads Programming. *https://computing.llnl.gov/tutorials/pthreads/*, 2017.

[7] Charles David Callahan. *A Global Approach to Detection of Parallelism*. PhD thesis, Rice University, 1987. https://hdl.handle.net/1911/16039.

[8] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[9] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. Parallel Programming in Futhark. https://buildmedia.readthedocs.org/media/pdf/futhark-book/latest/futhark-book.pdf, June 2019.

[10] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 27(4):662–731, 2005.

[11] William H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[13] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. Incremental flattening for nested data parallelism. In *Procs. Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 53–67. ACM, 2019.

[14] Troels Henriksen. *Design and Implementation of the Futhark Programming Language (Revised).* PhD thesis, University of Copenhagen, 2017.

[15] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*, FHPC'16, pages 38–43, New York, NY, USA, 2016. ACM.

[16] 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.

[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. *PLDI 2017*, 2017.

[18] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*. ACM Press, 2012.

[19] Sungdo Moon and Mary W. Hall. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*, pages 84–95, 1999.

[20] Nvidia Corporation, `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`. *NVIDIA Turing GPU Architecture - Graphics Reinvented.*

[21] C. E. Oancea, J. W. A. Selby, M. Giesbrecht, and S. M. Watt. Distributed Models of Thread-Level Speculation. In *Proceedings of the PDPTA'05*, pages 920–927, 2005.

[22] Cosmin E. Oancea and Alan Mycroft. Set-Congruence Dynamic Analysis for Software Thread-Level Speculation (TLS). In *Procs. Langs. Comp. Parallel Computing*, pages 156–171, 2008.

[23] Cosmin E. Oancea and Lawrence Rauchwerger. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Procs. International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 61–75, 2011.

[24] Cosmin E. Oancea and Lawrence Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 509–520, New York, NY, USA, 2012. ACM.

[25] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Symbolic range analysis of pointers. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization - CGO 2016*. ACM Press, 2016. `https://homepages.dcc.ufmg.br/~fernando/publications/papers/CGO16_paisante.pdf`.

[26] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. *ACM SIGPLAN Notices*, 30(6):67–78, June 1995.

[27] David J. Pearce. Integer Range Analysis for Whiley on Embedded Systems. In *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, April 2015. `http://homepages.ecs.vuw.ac.nz/~djp/files/SEUS15.pdf`.

[28] Marcin Pietron, Aleksander Byrski, and Marek Kisiel-Dorohinicki. Gpgpu for difficult black-box problems. In *Procedia Computer Science*, volume 51, pages 1023–1032, 2015.

[29] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. Parallel Distrib. System*, 10(2):160–199, 1999.

[30] Lawrence Rauchwerger, Nancy Amato, and David Padua. A Scalable Method for Run Time Loop Parallelization. *Int. Journal of Par. Prog*, 26:26–6, 1995.

[31] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation - PLDI '00*. ACM Press, 2000. `https://people.csail.mit.edu/rinard/paper/toplas05SymbolicBoundsAnalysis.pdf`.

[32] P. Rundberg and P. Stenström. An All-Software Thread-Level Data Dependence Speculation System for Multiprocs. *Journal of Instruction-Level Parallelism*, 1999.

[33] Silvius Rus, Jay Hoeflinger, and Lawrence Rauchwerger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *Int. Journal of Par. Prog*, 31(3):251–283, 2003.

[34] M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[35] Paraskevas Yiapanis, Gavin Brown, and Mikel Luján. Compiler-Driven Software Speculation for Thread-Level Parallelism. *ACM Transactions on Programming Languages and Systems*, 38(2):1–45, December 2015.

[36] Suan Hsi Yong and Susan Horwitz. Pointer-Range Analysis. In *Static Analysis*, pages 133–148. Springer Berlin Heidelberg, 2004. `https://research.cs.wisc.edu/wpis/papers/sas04.suan.pdf`.

Appendices

Appendix A Source Code Repository

The source code of the Futhark compiler can be found on GitHub at https://github.com/diku-dk/futhark where the variant with the husk operator can be found in the mgpu-husks branch at https://github.com/diku-dk/futhark/tree/mgpu-husks.

Appendix B Benchmarking Results

In this appendix we show the results of running subsets of the established Futhark benchmark programs, displaying the average of 100 runs of each data set. Notice that not all benchmark results show the exact data sets in use, but whenever it is relevant the actual data set will be specified.

B.1 Single GPU without the Husk Operator

The following table shows the benchmarking results for the established Futhark benchmark programs using a version of the Futhark compiler without the husk operator.

	Data	Avg. Runtime (µs)	RSD
accelerate/hashcat	rockyou.dataset	1272.190	0.300
accelerate/fluid	medium.in	1008.270	0.111
accelerate/fft	256x256.in	846.240	0.304
	128x512.in	848.960	0.278
	64x256.in	716.530	0.316
	512x512.in	1587.000	0.203
	1024x1024.in	7702.700	0.066
	128x128.in	738.770	0.539
accelerate/crystal	#1	84.450	0.432
	#5	3091.870	0.229
	#6	125013.148	0.009
	#7	10032.950	0.074
	#8	39996.289	0.020
	#9	199435.484	0.004
	#10	159648.734	0.006
	#11	249876.141	0.004

	Data	Avg. Runtime (µs)	RSD
accelerate/canny	lena256.in	174.000	0.294
	lena512.in	194.710	0.259
accelerate/tunnel	#1	439.290	0.257
	#2	785.450	0.274
	#3	2613.220	0.236
	#4	8540.320	0.102
	#5	33946.078	0.011
	#6	135567.750	0.007
	#7	271321.375	0.003
	#8	272307.969	0.006
	#9	513393.460	0.006
accelerate/smoothlife	#1	22776.449	0.025
	#2	28097.939	0.042
	#3	51820.078	0.011
	#4	308055.125	0.005
	#5	1284583.875	0.002
	#6	5482393.500	0.010
accelerate/trace	#1	746.800	0.230
	#2	50641.441	0.014
accelerate/pagerank	small.in	2029.770	0.161
	random_medium.in	5010.450	0.090

	Data	Avg. Runtime (µs)	RSD
accelerate/nbody	1000-bodies.in	161.840	0.132
	10000-bodies.in	1152.500	0.139
	100000-bodies.in	57318.430	0.022
	1000000-bodies.in	5278690.000	0.002
accelerate/mandelbrot	#1	207.430	0.175
	#2	254.020	0.153
	#3	815.060	0.253
	#4	2764.910	0.196
	#5	8312.910	0.081
	#6	32678.131	0.010
	#7	156756.469	0.008
	#8	1382882.375	0.001
	#9	128055.938	0.006
accelerate/kmeans	trivial.in	643.770	0.144
	k5_n50000.in	8095.810	0.069
	k5_n200000.in	13911.890	0.029
misc/bfast	sahara.in	13416.120	0.066
misc/bfast-cloudy	sahara-cloudy.in	21794.900	0.036
	peru.in	19329.039	0.027
jgf/series	10000.in	18045.070	0.041
	100000.in	166292.203	0.016
	1000000.in	1581907.000	0.005
jgf/keys	userkey0.txt	7801.740	0.063

	Data	Avg. Runtime (µs)	RSD
jgf/crypt	medium.in	512.970	0.264
fmpar/LocVolCalib	small.in	88350.398	0.025
	medium.in	67918.797	0.003
	large.in	764953.062	0.001
fmpar/OptionPricing	small.in	760.820	0.199
	medium.in	3540.940	0.156
	large.in	54695.781	0.017
misc/heston64	1062_quotes.in	59326.039	0.048
	10000_quotes.in	332948.156	0.021
	100000_quotes.in	3381069.000	0.020
misc/heston32	1062_quotes.in	22730.010	0.044
	10000_quotes.in	33686.199	0.041
	100000_quotes.in	210871.594	0.008
rodinia/bfs_iter_work_ok	4096nodes.in	2133.330	0.133
	512nodes_high_edge_variance.in	912.960	0.230
	graph1MW_6.in	6356.030	0.083
	64kn_32e-var-1-256-skew.in	2773.310	0.102

	Data	Avg. Runtime (µs)	RSD
rodinia/bfs_heuristic	4096nodes.in	1571.920	0.229
	512nodes_high_edge_variance.in	668.030	0.288
	graph1MW_6.in	5058.470	0.094
	64kn_32e-var-1-256-skew.in	2358.440	0.128
rodinia/bfs_filt_padded_fused	4096nodes.in	1374.670	0.176
	512nodes_high_edge_variance.in	576.570	0.239
	graph1MW_6.in	5028.380	0.094
	64kn_32e-var-1-256-skew.in	7228.770	0.093
rodinia/bfs_asympt_ok_but_slow	4096nodes.in	2338.350	0.179
	512nodes_high_edge_variance.in	935.830	0.186
	graph1MW_6.in	7980.530	0.040
	64kn_32e-var-1-256-skew.in	4707.000	0.108
rodinia/backprop	small.in	293.730	0.230
	medium.in	2960.520	0.182
	large.in	41306.012	0.019

	Data	Avg. Runtime (µs)	RSD
parboil/tpacf	small.in	4852.700	0.170
	medium.in	241980.734	0.004
	large.in	1303118.500	0.002
parboil/stencil	small.in	1873.030	0.084
	default.in	49928.219	0.019
parboil/sgemm	tiny.in	51.240	0.393
	small.in	60.040	0.292
	medium.in	2176.230	0.421
parboil/mri-q	small.in	1161.500	0.219
	large.in	5752.860	0.240
parboil/histo	default.in	280.930	0.245
	large.in	360.410	0.430
misc/radix_sort_large	radix_sort_10K.in	3321.020	0.183
	radix_sort_100K.in	3936.090	0.261
	radix_sort_1M.in	11722.110	0.052

	Data	Avg. Runtime (µs)	RSD
misc/radix_sort_blelloch_benchmark	radix_sort_10K.in	1097.300	0.195
	radix_sort_100K.in	1525.850	0.185
	radix_sort_1M.in	8914.580	0.102
Sune-ImageProc/interp_cos_plays	fake.in	11024.310	0.076
Sune-ImageProc/interp	fake.in	42240.410	0.022

	Data	Avg. Runtime (µs)	RSD
rodinia/srad	image.in	16420.721	0.031
rodinia/pathfinder	medium.in	1020.730	0.191
rodinia/particlefilter	128_128_10_image_10000_particles.in	7203.310	0.103
	128_128_10_image_400000_particles.in	140951.422	0.007
rodinia/nw	large.in	104635.398	0.011
rodinia/nn	medium.in	9184.760	0.062
rodinia/myocyte	small.in	152230.031	0.015
	medium.in	147303.484	0.010
rodinia/lud	16by16.in	278.050	0.212
	64.in	464.590	0.252
	256.in	2152.820	0.182
	512.in	4198.790	0.143
	2048.in	25906.850	0.058

	Data	Avg. Runtime (µs)	RSD
rodinia/lud-clean			
	16by16.in	197.340	0.361
	64.in	570.510	0.272
	256.in	2184.140	0.176
	512.in	4445.350	0.117
	2048.in	62874.289	0.035
rodinia/lavaMD			
	3_boxes.in	324.040	0.132
	10_boxes.in	2322.590	0.344
rodinia/kmeans			
	100.in	1707.970	0.104
	204800.in	77386.578	0.022
	kdd_cup.in	99789.820	0.010
rodinia/hotspot			
	64.in	1538.440	0.180
	512.in	4662.350	0.119
	1024.in	17711.689	0.041
rodinia/cfd			
	fvcorr.domn.097K.toa	1759428.375	0.004
	fvcorr.domn.193K.toa	3142535.750	0.010

B.2 Single GPU with the Husk Operator

The following table shows the benchmarking results for the established Futhark benchmark programs using a version of the Futhark compiler with the husk operator and using a single GPU.

	Data	Avg. Runtime (µs)	RSD
accelerate/hashcat	rockyou.dataset	1133.070	0.101
accelerate/fluid	medium.in	1014.830	0.039
accelerate/fft	256x256.in	903.760	0.006
	128x512.in	916.370	0.115
	64x256.in	691.120	0.026
	512x512.in	1785.930	0.015
	1024x1024.in	8087.530	0.027
	128x128.in	680.730	0.061
accelerate/crystal	#1	68.050	0.066
	#5	2703.510	0.104
	#6	114757.742	0.008
	#7	9188.990	0.021
	#8	36698.941	0.012
	#9	183454.719	0.005
	#10	146697.328	0.006
	#11	229705.234	0.005
accelerate/canny	lena256.in	152.780	0.029
	lena512.in	184.980	0.036

	Data	Avg. Runtime (µs)	RSD
accelerate/tunnel	#1	350.270	0.049
	#2	592.560	0.015
	#3	2352.390	0.117
	#4	7873.750	0.025
	#5	31266.711	0.012
	#6	124773.000	0.006
	#7	249784.484	0.005
	#8	249938.047	0.006
	#9	511468.820	0.005
accelerate/smoothlife	#1	22958.029	0.029
	#2	29951.520	0.043
	#3	57858.980	0.025
	#4	312850.750	0.007
	#5	1262049.875	0.002
	#6	5578220.000	0.001
accelerate/trace	#1	754.370	0.030
	#2	45904.160	0.039
accelerate/pagerank	small.in	1919.970	0.015
	random_medium.in	4938.440	0.039
accelerate/nbody	1000-bodies.in	204.770	0.018
	10000-bodies.in	1619.690	0.043
	100000-bodies.in	58912.551	0.012
	1000000-bodies.in	5326605.500	0.004

	Data	Avg. Runtime (µs)	RSD
accelerate/mandelbrot	#1	205.920	0.041
	#2	255.020	0.145
	#3	753.950	0.012
	#4	2403.320	0.077
	#5	7746.360	0.011
	#6	30183.510	0.011
	#7	144998.562	0.009
	#8	1278715.500	0.002
	#9	118524.531	0.008
accelerate/kmeans	trivial.in	674.560	0.032
	k5_n50000.in	7953.420	0.020
	k5_n200000.in	13695.700	0.064
misc/bfast	sahara.in	10682.250	0.062
misc/bfast-cloudy	sahara-cloudy.in	18514.670	0.044
	peru.in	17099.670	0.019
jgf/series	10000.in	16833.510	0.008
	100000.in	149116.953	0.007
	1000000.in	1406243.250	0.001
jgf/keys	userkey0.txt	7808.500	0.017
jgf/crypt	medium.in	525.410	0.023

	Data	Avg. Runtime (µs)	RSD
fmpar/LocVolCalib	small.in	86721.180	0.020
	medium.in	66455.344	0.015
	large.in	745787.188	0.003
fmpar/OptionPricing	small.in	1172.000	0.012
	medium.in	4139.080	0.032
	large.in	57923.012	0.007
misc/heston64	1062_quotes.in	58225.520	0.025
	10000_quotes.in	278777.031	0.006
	100000_quotes.in	2786435.500	0.001
misc/heston32	1062_quotes.in	23026.990	0.022
	10000_quotes.in	38519.949	0.013
	100000_quotes.in	196144.422	0.008
rodinia/bfs_iter_work_ok	4096nodes.in	2119.220	0.039
	512nodes_high_edge_variance.in	860.550	0.072
	graph1MW_6.in	6312.770	0.037
	64kn_32e-var-1-256-skew.in	2759.940	0.040

	Data	Avg. Runtime (µs)	RSD
rodinia/bfs_heuristic	4096nodes.in	1371.290	0.024
	512nodes_high_edge_variance.in	628.580	0.017
	graph1MW_6.in	4876.480	0.038
	64kn_32e-var-1-256-skew.in	2295.130	0.021
rodinia/bfs_filt_padded_fused	4096nodes.in	1398.450	0.337
	512nodes_high_edge_variance.in	572.180	0.061
	graph1MW_6.in	4837.880	0.065
	64kn_32e-var-1-256-skew.in	6790.400	0.019
rodinia/bfs_asympt_ok_but_slow	4096nodes.in	2096.170	0.018
	512nodes_high_edge_variance.in	890.920	0.014
	graph1MW_6.in	7568.340	0.056
	64kn_32e-var-1-256-skew.in	4446.520	0.027
rodinia/backprop	small.in	298.400	0.036
	medium.in	4944.030	0.019
	large.in	73698.711	0.011

	Data	Avg. Runtime (µs)	RSD
parboil/tpacf	small.in	4411.640	0.102
	medium.in	223328.484	0.007
	large.in	1181783.000	0.003
parboil/stencil	small.in	2866.890	0.011
	default.in	48325.621	0.026
parboil/sgemm	tiny.in	49.020	0.038
	small.in	53.380	0.123
	medium.in	2656.450	0.104
parboil/mri-q	small.in	1270.930	0.005
	large.in	5991.630	0.101
parboil/histo	default.in	255.270	0.014
	large.in	306.250	0.020
misc/radix_sort_large	radix_sort_10K.in	3096.150	0.037
	radix_sort_100K.in	3606.610	0.034
	radix_sort_1M.in	11392.220	0.086

	Data	Avg. Runtime (µs)	RSD
misc/radix_sort_blelloch_benchmark	radix_sort_10K.in	1046.700	0.020
	radix_sort_100K.in	1455.820	0.054
	radix_sort_1M.in	8313.680	0.064
Sune-ImageProc/interp_cos_plays	fake.in	10149.240	0.015
Sune-ImageProc/interp	fake.in	38082.051	0.009

	Data	Avg. Runtime (µs)	RSD
rodinia/srad	image.in	17083.039	0.040
rodinia/pathfinder	medium.in	959.920	0.013
rodinia/particlefilter	128_128_10_image_10000_particles.in	7362.870	0.081
	128_128_10_image_400000_particles.in	133587.047	0.010
rodinia/nw	large.in	99654.422	0.021
rodinia/nn	medium.in	20672.711	0.067
rodinia/myocyte	small.in	139555.875	0.013
	medium.in	148814.297	0.006
rodinia/lud	16by16.in	374.270	0.041
	64.in	731.020	0.016
	256.in	3030.900	0.025
	512.in	6660.230	0.088
	2048.in	89245.406	0.025

	Data	Avg. Runtime (µs)	RSD
rodinia/lud-clean			
	16by16.in	233.430	0.046
	64.in	804.370	0.053
	256.in	3078.090	0.051
	512.in	6008.160	0.019
	2048.in	31894.730	0.036
rodinia/lavaMD			
	3_boxes.in	278.090	0.010
	10_boxes.in	1569.590	0.030
rodinia/kmeans			
	100.in	1890.900	0.011
	204800.in	78111.000	0.023
	kdd_cup.in	104191.930	0.010
rodinia/hotspot			
	64.in	1445.090	0.028
	512.in	4507.450	0.111
	1024.in	17198.881	0.053
rodinia/cfd			
	fvcorr.domn.097K.toa	1187663.875	0.010
	fvcorr.domn.193K.toa	1774197.750	0.004

B.3 Multiple GPUs with the Husk Operator

The following table shows the benchmarking results for the established Futhark benchmark programs using a version of the Futhark compiler with the husk operator and using a two GPUs.

	Data	Avg. Runtime (µs)	RSD
finpar/LocVolCalib	small.in	88224.219	0.094
	medium.in	51178.730	0.109
	large.in	399335.906	0.139
accelerate/tunnel	#1	342.460	0.029
	#2	585.670	0.024
	#3	1725.940	0.049
	#4	5357.360	0.016
	#5	20770.920	0.011
	#6	83034.828	0.004
	#7	166037.625	0.003
	#8	166310.328	0.003
	#9	277573.240	0.007
accelerate/mandelbrot	#1	332.640	0.148
	#2	356.430	0.036
	#3	865.870	0.026
	#4	2687.040	0.028
	#5	9235.570	0.008
	#6	35808.031	0.006
	#7	92670.070	0.004
	#8	661512.750	0.002
	#9	142400.938	0.003
rodinia/cfd	fvcorr.domn.097K.toa	5471266.000	0.022
	fvcorr.domn.193K.toa	7837625.000	0.003

Appendix C Profiling Results

All the following appendices are summarized results from running the corresponding Futhark programs monitored by the NVIDIA CUDA profiling tool.

C.1 Single GPU without the Husk Operator

The profiling results in this appendix are from Futhark programs compiled with the current release of the Futhark compiler using the CUDA C backend.

C.1.1 Rodinia NN

Table 9 shows a summary of the results of using the NVIDIA CUDA profiling tool to run the Rodinia-based NN benchmark program from the Futhark benchmark collection, using the supplied medium.in data set.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
57.691	1.648	103	0.016	[CUDA memcpy HtoD]
31.612	0.892	100	0.009	segred_nonseg_5121
7.711	0.217	200	0.001	[CUDA memcpy DtoD]
1.994	0.056	101	0.001	[CUDA memcpy DtoH]
0.899	0.025	1	0.025	segmap_5087
0.048	0.001	1	0.001	replicate_5167
0.044	0.001	1	0.001	replicate_5172

Table 9: NVIDIA CUDA profiler results for the NN benchmark program without the husk operator, run on the `medium.in` data set.

C.1.2 Rodinia CFD

Table 10shows a summary of the results of using the NVIDIA CUDA profiling tool to run the Rodinia-based CFD benchmark program from the Futhark benchmark collection, using the supplied fvcorr.domn.193K.toa.gz data set.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
32.667	726.577	12000	0.061	map_transpose_f32_low_width
30.564	679.809	12000	0.057	map_transpose_f32
25.038	556.886	4000	0.139	segmap_22193
11.578	257.522	2000	0.129	segmap_21938
0.127	5.666	7	0.809	[CUDA memcpy HtoD]
0.025	1.116	1	1.116	[CUDA memcpy DtoH]
0.001	0.011	1	0.011	segmap_21927
0.000	0.002	1	0.002	[CUDA memcpy DtoD]

Table 10: NVIDIA CUDA profiler results for the CFD benchmark program without the husk operator, run on the `fvcorr.domn.193K.toa.gz` data set.

C.2 Single GPU with the Husk Operator

The profiling results in this appendix are from Futhark programs compiled with the new Futhark compiler with the husk operator, using the CUDA C backend. These results are for running the benchmark programs using a singe GPU.

C.2.1 Finpar LocVolCalib

Table 11 shows a summary of the results of using the NVIDIA CUDA profiling tool to run the local volatility calibration benchmark program from the parallel financial part of the Futhark benchmark collection, using the supplied large.in data set.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
34.325	260.845	694	0.376	map_transpose_f32
21.472	163.172	63	2.590	segmap_45887
16.246	123.461	63	1.960	segmap_45746
7.461	56.697	63	0.900	segmap_45810
6.724	51.100	63	0.811	copy_46746
5.775	43.883	63	0.697	segmap_46046
4.872	37.022	63	0.588	segmap_45952
3.076	23.379	63	0.371	copy_46763
0.025	0.192	1	0.192	segmap_46133
0.022	0.168	63	0.003	segmap_45855
0.000	0.003	2	0.001	map_transpose_f32_low_height
0.000	0.002	1	0.002	segmap_45562
0.000	0.002	1	0.002	segmap_45600
0.000	0.002	1	0.002	segmap_45652
0.000	0.002	1	0.002	segmap_46155
0.000	0.002	1	0.002	copy_46768
0.000	0.001	1	0.001	segmap_45581
0.000	0.001	1	0.002	[CUDA memcpy DtoH]

Table 11: NVIDIA CUDA profiler results for the LocVolCalib benchmark program with the husk operator, run on the `large.in` data set using a single GPU.

C.2.2 Accelerate Tunnel

Table 12 and 13 shows summaries of the results of using the NVIDIA CUDA profiling tool to run the Tunnel benchmark program from the Futhark benchmark collection, using the input "`10f32 16000 32000`" and "`100f32 16000 16000`" respectively.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
99.999	250.920	1	250.920	segmap_12705
0.001	0.002	2	0.001	[CUDA memcpy DtoH]

Table 12: NVIDIA CUDA profiler results for the Tunnel benchmark program with the husk operator, run with the input "`10f32 16000 32000`" using a single GPU.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
99.999	521.285	1	521.285	segmap_12705
0.001	0.003	2	0.002	[CUDA memcpy DtoH]

Table 13: NVIDIA CUDA profiler results for the Tunnel benchmark program with the husk operator, run with the input "`100f32 16000 16000`" using a single GPU.

C.2.3 Accelerate Mandelbrot

Table 14 and 15 show summaries of the results of using the NVIDIA CUDA profiling tool to run the Mandelbrot benchmark program from the Futhark benchmark collection, using the input "`16000 16000 -0.7f32 0f32 3.067f32 10000 16f32`" and "`32000 32000 -0.7f32 0f32 3.067f32 100 16f32`" respectively.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
100.000	1288.013	1	1288.013	segmap_26026
0.000	0.003	2	0.001	[CUDA memcpy DtoH]

Table 14: NVIDIA CUDA profiler results for the Mandelbrot benchmark program with the husk operator, run with the input "`16000 16000 -0.7f32 0f32 3.067f32 10000 16f32`" using a single GPU.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
99.998	238.457	1	238.457	segmap_26026
0.002	0.003	2	0.001	[CUDA memcpy DtoH]

Table 15: NVIDIA CUDA profiler results for the Mandelbrot benchmark program with the husk operator, run with the input "`32000 32000 -0.7f32 0f32 3.067f32 100 16f32`" using a single GPU.

C.2.4 Rodinia NN

Table 16 shows a summary of the results of using the NVIDIA CUDA profiling tool to run the Rodinia-based NN benchmark program from the Futhark benchmark collection, using the supplied medium.in data set.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
62.676	1.767	303	0.003	[CUDA memcpy HtoD]
32.395	0.913	100	0.009	segred_nonseg_5152
3.978	0.112	201	0.001	[CUDA memcpy DtoH]
0.856	0.024	1	0.024	segmap_5095
0.051	0.001	1	0.001	replicate_5204
0.044	0.001	1	0.001	replicate_5209

Table 16: NVIDIA CUDA profiler results for the NN benchmark program with the husk operator, run on the `medium.in` data set using a single GPU.

C.2.5 Rodinia CFD

Table 17 shows a summary of the results of using the NVIDIA CUDA profiling tool to run the Rodinia-based CFD benchmark program from the Futhark benchmark collection, using the supplied fvcorr.domn.193K.toa.gz data set.

Time (%)	Time (ms)	Calls	Avg (ms)	Name
28.588	392.628	4000	0.098	segmap_22235
14.066	193.189	2000	0.097	segmap_21962
12.441	170.861	6000	0.028	map_transpose_f32_low_height
12.114	166.381	6000	0.028	map_transpose_f32_low_width
5.802	79.686	4000	0.020	copy_22692
5.475	75.192	4000	0.019	copy_22707
5.363	73.654	4000	0.018	copy_22702
5.260	72.238	4000	0.018	copy_22697
2.801	38.465	2000	0.019	copy_22666
2.727	37.457	2000	0.019	copy_22656
2.690	36.945	2000	0.018	copy_22661
2.419	33.222	2000	0.017	copy_22651
0.210	5.781	7	0.826	[CUDA memcpy HtoD]
0.041	1.168	1	1.168	[CUDA memcpy DtoH]
0.001	0.011	1	0.011	segmap_21933
0.001	0.002	1	0.002	[CUDA memcpy DtoD]

Table 17: NVIDIA CUDA profiler results for the CFD benchmark program with the husk operator, run on the `fvcorr.domn.193K.toa.gz` data set using a single GPU.

C.3 Multiple GPUs with the Husk Operator

The profiling results in this appendix are from Futhark programs compiled with the new Futhark compiler with the husk operator, using the CUDA C backend. These results are for running the benchmark programs using two GPUs.

C.3.1 Finpar LocVolCalib

Table 18 shows a summary of the results of using the NVIDIA CUDA profiling tool to run the local volatility calibration benchmark program from the parallel financial part of the Futhark benchmark collection, using the supplied large.in data set and two GPUs.

	Time (%)	Time (ms)	Calls	Avg (ms)	Name
GPU 0	35.411	134.074	694	0.193	map_transpose_f32
	20.439	77.386	63	1.228	segmap_45887
	15.897	60.190	63	0.955	segmap_45746
	7.449	28.205	63	0.448	segmap_45810
	6.973	26.403	63	0.419	copy_46746
	5.680	21.505	63	0.341	segmap_46046
	5.044	19.096	63	0.303	segmap_45952
	3.034	11.489	63	0.182	copy_46763
	0.038	0.142	63	0.002	segmap_45855
	0.027	0.102	1	0.102	segmap_46133
	0.003	0.013	9	0.001	[CUDA memcpy PtoP]
	0.001	0.003	2	0.002	map_transpose_f32_low_height
	0.001	0.002	1	0.002	segmap_45600
	0.001	0.002	1	0.002	segmap_45652
	0.001	0.002	1	0.002	segmap_46155
	0.000	0.002	1	0.002	segmap_45562
	0.000	0.002	1	0.002	copy_46768
	0.000	0.002	1	0.002	[CUDA memcpy DtoH]
	0.000	0.001	1	0.001	segmap_45581
GPU 1	35.193	132.362	694	0.191	map_transpose_f32
	20.640	77.629	63	1.232	segmap_45887
	15.973	60.075	63	0.954	segmap_45746
	7.547	28.386	63	0.451	segmap_45810
	6.825	25.668	63	0.407	copy_46746
	5.592	21.030	63	0.334	segmap_46046
	5.208	19.588	63	0.311	segmap_45952
	2.952	11.104	63	0.176	copy_46763
	0.037	0.191	63	0.002	segmap_45855
	0.027	0.102	1	0.102	segmap_46133
	0.001	0.006	2	0.002	map_transpose_f32_low_height
	0.001	0.002	1	0.002	segmap_45600
	0.001	0.002	1	0.002	segmap_46155
	0.001	0.002	1	0.002	segmap_45652
	0.000	0.002	1	0.002	copy_46768
	0.000	0.002	1	0.002	segmap_45562
	0.000	0.001	1	0.001	segmap_45581

Table 18: NVIDIA CUDA profiler results for the LocVolCalib benchmark program with the husk operator, run on the `large.in` data set using two GPUs.

C.3.2 Accelerate Tunnel

Table 19 and 20 show summaries of the results of using the NVIDIA CUDA profiling tool to run the Tunnel benchmark program from the Futhark benchmark collection, using the input "`10f32 16000 32000`" and "`100f32 16000 16000`"

respectively, both using two GPUs.

	Time (%)	Time (ms)	Calls	Avg (ms)	Name
GPU 0	76.429	136.848	1	136.848	segmap_12705
	23.570	42.203	1	42.203	[CUDA memcpy PtoP]
	0.002	0.002	2	0.001	[CUDA memcpy DtoH]
GPU 1	100.000	143.781	1	143.781	segmap_12705

Table 19: NVIDIA CUDA profiler results for the Tunnel benchmark program with the husk operator, run with the input "`10f32 16000 32000`" using two GPUs.

	Time (%)	Time (ms)	Calls	Avg (ms)	Name
GPU 0	92.344	254.574	1	254.574	segmap_12705
	7.655	21.103	1	21.103	[CUDA memcpy PtoP]
	0.001	0.003	2	0.002	[CUDA memcpy DtoH]
GPU 1	100.000	270.330	1	270.330	segmap_12705

Table 20: NVIDIA CUDA profiler results for the Tunnel benchmark program with the husk operator, run with the input "`10f32 16000 32000`" using two GPUs.

C.3.3 Accelerate Mandelbrot

Table 21 and 22 show summaries of the results of using the NVIDIA CUDA profiling tool to run the Mandelbrot benchmark program from the Futhark benchmark collection, using the input "`16000 16000 -0.7f32 0f32 3.067f32 10000 16f32`" and "`32000 32000 -0.7f32 0f32 3.067f32 100 16f32`" respectively, both using two GPUs.

	Time (%)	Time (ms)	Calls	Avg (ms)	Name
GPU 0	96.950	670.821	1	670.821	segmap_26026
	3.050	21.103	1	21.103	[CUDA memcpy PtoP]
	0.000	0.003	2	0.002	[CUDA memcpy DtoH]
GPU 1	100.000	648.760	1	648.760	segmap_26026

Table 21: NVIDIA CUDA profiler results for the Mandelbrot benchmark program with the husk operator, run with the input "`16000 16000 -0.7f32 0f32 3.067f32 10000 16f32`" using two GPUs.

	Time (%)	Time (ms)	Calls	Avg (ms)	Name
GPU 0	59.319	84.590	1	84.590	[CUDA memcpy PtoP]
	40.679	58.008	1	58.008	segmap_26026
	0.002	0.002	2	0.001	[CUDA memcpy DtoH]
GPU 1					
	100.000	62.562	1	62.562	segmap_26026

Table 22: NVIDIA CUDA profiler results for the Mandelbrot benchmark program with the husk operator, run with the input "`32000 32000 -0.7f32 0f32 3.067f32 100 16f32`" using two GPUs.

C.3.4 Rodinia CFD

Table 23 shows a summary of the results of using the NVIDIA CUDA profiling tool to run the Rodinia-based CFD benchmark program from the Futhark benchmark collection, using the supplied fvcorr.domn.193K.toa.gz data set and two GPUs.

	Time (%)	Time (ms)	Calls	Avg (ms)	Name
GPU 0	88.226	5303.487	36002	0.147	[CUDA memcpy PtoP]
	3.129	188.121	4000	0.047	segmap_22235
	2.323	139.635	6000	0.023	map_transpose_f32_low_width
	1.460	87.746	2000	0.044	segmap_21962
	1.205	72.415	6000	0.012	map_transpose_f32_low_height
	0.665	39.985	4000	0.010	copy_22692
	0.612	36.772	4000	0.009	copy_22707
	0.595	35.791	4000	0.009	copy_22697
	0.564	33.925	4000	0.008	copy_22702
	0.314	18.872	2000	0.009	copy_22666
	0.308	18.504	2000	0.009	copy_22656
	0.283	16.987	2000	0.008	copy_22661
	0.255	15.342	2000	0.008	copy_22651
	0.050	6.013	7	0.859	[CUDA memcpy HtoD]
	0.011	1.271	1	1.271	[CUDA memcpy DtoH]
	0.000	0.006	1	0.006	segmap_21933
	0.000	0.002	1	0.002	[CUDA memcpy DtoD]
GPU 1	32.990	209.572	4000	0.052	segmap_22235
	15.454	98.169	2000	0.049	segmap_21962
	13.991	88.880	6000	0.015	map_transpose_f32_low_height
	6.350	40.337	4000	0.010	copy_22692
	6.281	39.897	4000	0.010	copy_22697
	6.240	39.642	4000	0.010	copy_22702
	6.187	39.303	4000	0.010	copy_22707
	3.168	20.122	2000	0.010	copy_22651
	3.141	19.955	2000	0.010	copy_22661
	3.137	19.924	2000	0.010	copy_22666
	3.060	19.441	2000	0.010	copy_22656

Table 23: NVIDIA CUDA profiler results for the CFD benchmark program with the husk operator, run on the fvcorr.domn.193K.toa.gz data set using two GPUs.