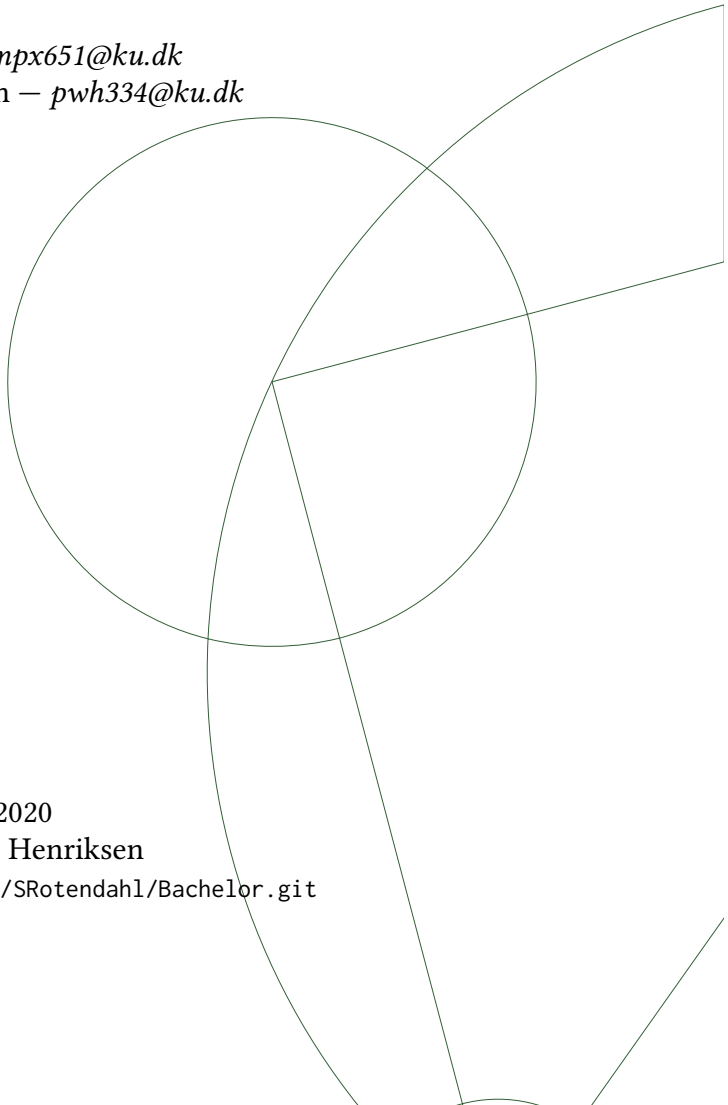
UNIVERSITY OF COPENHAGEN

BSc Thesis

Autotuning Futhark

Simon Rotendahl — *mpx651@ku.dk*
Carl Mathias Graae Larsen — *pwh334@ku.dk*



January 9, 2020
Supervisor: Troels Henriksen
Source code: `https://github.com/SRotendahl/Bachelor.git`

Contents

Abstract

This report describes the implementation of an autotuner for the programming language Futhark. This particular autotuner is implemented to test the notion of Futhark programs being small enough to be comfortably autotuned by exhaustively searching all executions for the best execution.

The autotuner is shown to work, always picking the best execution, achieving executions up to 26.97 times faster, while in some cases giving the same result compared to not tuning, as well as slightly outperforming Futharks existing autotuner.

However the exhaustive autotuner is slow at autotuning. We see that the amount of datasets and the variance in them, is crucial for the amount of time spend autotuning. The autotuner performed very well if there was only a single dataset, but for large programs with multiple dataset it took 23 hours to autotune.

We conclude that an exhaustive autotuner is not suitable, if the underlying concept for tuning is to add a multitude of fairly random datasets and tune the parameters from that. However if it turns out to be possible to create more representative datasets, that can reduce the number of datasets needed to autotune a program then an exhaustive autotuner might be feasible.

1 Introduction

GPGPU (General-purpose computing on graphics processing units) have been on the rise, due to limitations on CPUs, such as power consumption, and the fact that the computational potential of GPUs (Graphical processing unit) is vastly superior to CPUs, as seen in Figure 1. There are technical and physical issue in reaching this potential, for example memory bandwidth bottle-necking computation, but there is also the conceptual issue of how to map parallelism to parallel hardware.

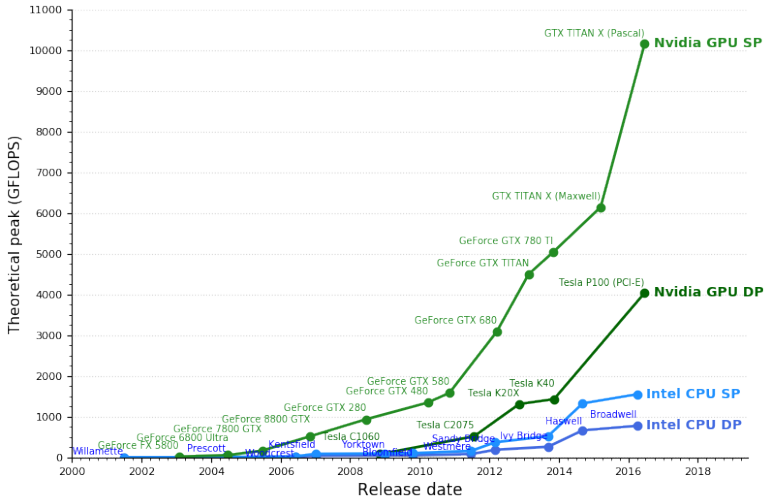


Figure 1: A graph showing the computational potential of CPUs and GPUs from 2000—2016 [4]

In this project we will implement an autotuner for the programming language Futhark [6], which is an ongoing research project at The University of Copenhagen. Futhark is a statically typed, data-parallel, and purely functional array language. The aim of Futhark is to shift the burden of writing efficient parallel code from the programmer to the compiler. The burden referred to is the requirement for a deep understanding of GPU hardware architecture and compiler knowledge for the language used, which is necessary to write efficient GPU code. Even when this requirements is met, using common GPU languages such as CUDA and OpenCL will still result in non-portable code as a lot of the optimizations made are specific to a certain GPU, or at least to a certain GPU architecture.

A Futhark program for matrix-matrix multiplication can be seen in listing 1. The syntax is similar to languages such as ML and Haskell. It is a good example of how Futhark differs from CUDA or OpenCL (we would have included an example of CUDA, but it was too long, it can be found in Appendix A.1). It should be clear why, if Futhark has the possibility to achieve speeds similar to those of CUDA and OpenCL Futhark would be very preferable.

To achieve the shift in burden from programmer to compiler, the Futhark compiler creates

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

let main [n][m][p] (xss: [n][m]f32) (yss: [m][p]f32): [n][p]f32 =
map (\xs -> map (dotprod xs) (transpose yss)) xss
```

Listing 1: Matrix-matrix multiplication in Futhark [8]

multiple semantically equivalent parallelisations, called code versions, for each parallel construct. The code versions are combined into a single program where the semantically equivalent code versions are guarded by a run-time comparison, so that only a single one of them are executed and the semantics of the new program is equivalent to the original Futhark program. The comparisons are between a user defined parameters and values that are a reflection of the input size.

Adjusting these parameters to select the execution that will minimize the running time is called tuning. The aim of this project is to automatize this process, so that it does not have to be done by hand. This is what we call autotuning. To find the optimal set of values for the parameters our autotuner is going to perform an exhaustive search, that means trying out every meaningful set of values and select the best one. We perform an exhaustive search motivated by the suspicion that Futhark programs are relatively small and therefore an exhaustive search might be a feasible solution.

The remainder of this report is structured as follows. In section 2 we present what parallelism is and the concepts behind how Futhark transforms functional code into efficient parallel code. We also present what it means to autotune a Futhark program. In section 3 we explore how we have implemented the exhaustive autotuner along with thoughts about future concerns the autotuner may face due to Futhark being an ongoing project. We evaluate the autotuner in section 4 exploring the feasibility of the exhaustive autotuner. We have then reflected on the results achieve, and suggested improvements in section 5.

2 Background

In this section, we want to present the theory and concepts necessary, to understand how to autotune Futhark programs. In order to grasp these concepts, a simplified explanation of the hardware we execute parallel code on, that being GPUs, is in order.

The idea of a GPU is to devote more transistors to compute, than a CPU, with the consequence of less transistors for data caching and flow control. More specifically, a GPU contains a high number of *streaming processors (SPs)*. SPs are general purpose, but have significant limitations, like not supporting branch prediction, however this is made up for by instruction-level parallelism. A thread is an abstraction of a single execution of some function. An SP is capable of executing a large number of threads at once while being able to efficiently switch context. We say that on an SP, there can fit one workgroup, so if an SP can execute 128 threads, we say that SP (and therefore that GPU) has a maximum

workgroup size of 128. Since all the threads in a workgroup are running on the same SP they are able to communicate through the SPs internal memory, called *shared memory*, which is much faster than having to use the GPUs "main" memory, called *global memory*. Because using the shared memory is fast, while using the global memory is slow, splitting work across SPs (workgroups) and handle their intermediate results can be expensive.

2.1 Parallelism

In the real world there are limitations to parallel hardware. We commonly execute parallel work on GPUs, due to its high number of threads. The limitation on GPUs, most essential to this report, is that a thread, may not spawn additional threads. [1] We say that since a GPU thread cannot spawn additional threads, it only supports *flat* parallelism. In the following sections approaches to work within this limitation are explored.

2.2 Flat parallelism

Many problems do not have a flat structure, as our current platform effectively requires. Imagine a function (`drawline`) that, given two endpoints, will determine all the pixel that lie in a straight line in between. This function will have a flat structure, and can therefore be executed in parallel. The function cannot itself be drawn in parallel as this would be nested parallelism, which is not supported. So while a user would be able to draw a single line in parallel with this function, it would not be possible to use it to draw multiple lines in parallel.

Therefore we need to map nested parallelism to a flat structure. This problem has already been conceptually solved by *flattening*, and implemented in the language NESL [3].

When flattening a datastructure, we want to keep the information regarding the original nest. In the original work presenting flattening [2] this is done with a structure called *field*. A *field* has the form of `(nest information, value)`, where a value can be an additional field, allowing for arbitrary depth. For a multidimensional array the field would be `(segment lengths, values/fields)`, below here is a concrete example.

`[([1,2,3], [4,5], [5,7,8,9])]` $\rightarrow$ `([3], ([3,2,4], [1,2,3,4,5,6,7,8,9]))`

The length of the outer array is 3, and since there is only a single segment that is the only lenght we save. In the inner *field* we need to save the length of all three of the segments, and all the values of the original subarrays.

Flattening allows for completely flat data that can be mapped directly to the GPU. But, as we saw in the example above, flattening increases the memory cost and it does not take communication cost into account. This is a problem as GPUs are not infinitely parallel, they have some maximum physical capacity for parallelism. Going beyond that capacity will not result in futher computational gain and the flattening has rendered certain optimizations, like exploiting locality of reference, impossible.

[1] While this is not strictly true, in practice it remains so.

2.3 Moderate flattening

To determine the amount of parallelism to exploit, we need a closer loke at GPUs. *compute kernels*, simply refered to as *kernels* in this report, are the routines that are run on parallel hardware such as GPUs. In low level languages such as `OpenCl C` a kernel is a function the programmer defines. The function is then executed $N \times M$ times in parallel by N different *workgroups* each with M *threads*. The matrix-matrix multiplication in appendix A.1 is an example of a kernel in the `CUDA` language.

A kernel can also be thought of as the work, that is to be done, in a perfect *parallel nest*. A perfect parallel nest is a nest where all the work, is at the innermost level. Some contrived examples of perfect and imperfect nests can be seen in Figure 2.

```
1 map (map (*2)) arr
```

(a) A perfect `map` nest in Haskell, multiplying every element of a two dimensional array by 2.

```
1 map (\x -> let y = 2:x in map (*2) y)
```

(b) An imperfect `map` nest in Haskell, prepending 2 to every array x, and multiplying every element of each x by 2.

```
1 map (scanl (+) 0)
```

(c) A perfect `map-scan` nest in Haskell, left-scanning every element of a two dimensional array.

```
1 map (\x -> if (head) x > 3 then scanl (+) 0 x else x)
```

(d) A imperfect `map-scan` nest in Haskell, which only scans an element of the array, if the first element of the nested array is over 3.

Figure 2: Two examples of perfect and imperfect code. In 2a and 2b we see two `map` nests, where 2b is not perfectly nested due to the concatenation being performed in between. In the two `map-scan` nests of 2c and 2d we see that one is perfectly nested, while the other has a conditional expression in between making it imperfectly nested.

```
1 map (\xs -> let y = reduce (+) 0 xs
2          in map (+y) xs)
3      xss
```

(a) Code fragment before distribution.

```
1 let ys = map (\xs -> reduce (+) 0 xs) xss
2 in map (\xs y -> map (+y) xs) xss ys
```

(b) Code fragment after distribution.

Figure 3: Two version of a fragment of code, showing the distribution of SOACs (*second-order array combinators*) [5].

Futhark transforms nested data-parallelism into flat structures from which kernels can later be extracted. An example of this transformation can be seen in figure 3. We can see that the outer `map`, of figure 3a, contains more parallelism in it, in the form of the parallel constructs `reduce` and `map`. So in the case where the outer `map` does not saturate the GPU's capacity by itself we want to exploit the parallelism of the `map`s body. With figure 3a this is not possible, since the outer `map` would be parallelised and the inner sequentialized to one GPU kernel. Instead we can transform it to figure 3b. Here the parallel constructs in the `map`s body are

distributed out into their own `map` nests, giving two perfect `map` nests. Each line in 3b can then be translated into a GPU kernel, the result from the first being used in the second.

This transformation is dubbed *moderate flattening* [9] due to its conceptual resemblance to the flattening algorithm put forth by Blelloch and Sabot [2]. However it is moderate in its approach only flattening the parallel construct as long as the parallelism does not oversaturate the hardware, otherwise it efficiently sequentialize the remaining parallelism. As we saw in Figure 3 Futhark flattens nested parallel construct by extracting kernels based on certain rewrite/flattening rules[9]. At the time of the implementation of moderate flattening the algorithm that applied these rules was based on heuristics about the structure of `map` nest contexts. These where decent approximations but because of the nature of GPU's (having different capacities for parallelism) there is no one size fits all.

It is important to note that Futhark does not at the time of this project support irregular data structures. This means that we cannot have a multidimensional array where the elements have different shapes. The reason for this limitation is that allowing irregular data structures introduces a time, and memory, overhead that can be hard for the programmer to understand or reason about. Making the programmer to flatten their datastructure by hand forces them to consider, and understand, this overhead.[7]

2.4 Incremental flattening

While moderate flattening was a good start it does not differentiate between different hardware capabilities and size of the data being worked on. This is where incremental flattening comes in. In short incremental flattening statically generates multiple different, semantically equivalent, piecewise code versions of the same program based on the rules of moderate flattening (along with some additional ones) [10]. It's then dynamically decided at run-time whether to further exploit the parallelism or not. To get an intuition for

```
1 let dotprod [n] (xs: [n]f32) (ys: [n]f32): f32 =
2 reduce (+) 0f32 (map2 (*) xs ys)
3
4 let main [n][m][p] (xss: [n][m]f32) (yss: [m][p]f32): [n][p]f32 =
5 map (\xs -> map (dotprod xs) (transpose yss)) xss
```

Figure 4: Matrix-matrix multiplication in Futhark [8]

incremental flattening lets go back to the example of matrix-matrix multiplication, which can be seen in Figure 4. As matrix-matrix multiplication contains a lot of `map`s (as is common in GPU programs do) we will briefly explain the code versions generated by the flattening rule when a `map` containing additional parallelism, is encountered (CV is short for code version).

CV0 The body of the map will be executed sequentially. This means that there will be assigned one GPU thread to each element of the array being mapped over, each thread executing the map body sequentially.

CV1 The body of the map is partially executed in parallel. This means that there will be assigned one GPU workgroup to each element of the array being mapped over. This is partial since it does not exploit the parallelism fully as there will most likely not be enough threads to fully parallelise the entire construct within the `map` body.

$\cdots$ Continues to flatten the `map` function, since we still have further parallel capacity to exploit.

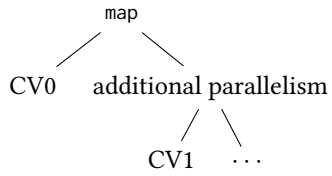


Figure 5: A tree showing the structure of the three different code versions generate by flattening a `map` nest

In matrix-matrix multiplication, there are three levels of nested parallelism to exploit: the outer `map`, the inner `map` and the `dotprod` function. Lets look at the different code versions Futhark generates, each code version denoted by a V [10].

V0) The outer `map` is distributed out across the parallel construct executing its body sequentially. More specifically one thread calculates one row in the resulting matrix each. This is CV0 for `map`.

V1) The outer `map` is executed in parallel, and the inner `map` of `main` is executed partially in parallel. More specifically one workgroup calculates one row in the resulting matrix. This is CV1 for `map`.

V2) The two outer maps of `main` are executed in parallel and their body (`dotprod`) is executed sequentially. More specifically each thread is calculating one element in the resulting matrix. This is CV0 for `map`.

V3) The two outer `maps`, does not saturate the GPU, but the `dotprod` function would oversaturate it, therefore, `dotprod` is partially executed in parallel. More specifically each element of the result matrix is executed by a workgroup. This is CV1

V4) If the entire parallel nest does not oversaturate the GPU, the entire nest is executed in parallel. This means there is a thread for each multiplication in the `dotprod` functions.

The optimal choice for which of these semantically equivalent versions gets executed depends on hardware and the input data. So each version is guarded by a choice that depends on these factors. To visualize the choices, we will look at them as a tree. In Figure 6, we see that the matrix multiplication program has four choices that are dependent on each other.

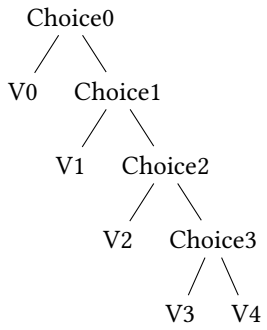


Figure 6: The structure of choices found in the futhark program for matrix-matrix multiplication (see listing 4). (V) represents the resulting code version of a choice

With these dependent choices, the idea of incremental flattening becomes increasingly clear. We step through the code, and decided whether we should flatten the parallelism available, or whether we should sequentialize it and exploit locality-of-reference optimizations. Due to the dependency of the choices, we end up iterating through the parallel nest, and flattening until we reach the full capacity of the hardware, hence the name *incremental flattening*. The choices will not always be in form of a unbalanced tree (although they are for the most part), they can also be balanced, and/or form a forest, we will give an example of a more complicated tree later.

2.5 Structure of the code versions

To further understand how we will choose the correct code version, we have filled out the matrix-matrix multiplication tree, with more detail, which can be seen in Figure 7. In the tree the non-leaf nodes are the statically generated predicates (the choice), hence forth known as *threshold comparisons* or just *comparisons*. the comparisons guard the different code versions, ensuring the sematics of the program does not change. The comparisons have a *threshold parameter* on the left t_i, which is a symbolic representation of the parallel capacity of the hardware. On the right we have a value that reflects the shape and size of the data, as seen in the figure.

To inspect the structure of these predicates and thresholds parameters further, lets look at a more complex Futhark program, from the Futhark benchmarks, called *LocVolCalib*. It is important not to get an end node (code version) confused with a complete program. While this can be the case, we will give two examples of paths, and corresponding code versions, through the tree in Figure 8, to illustrate this;

- `{(Th4, False), (Th5, False), (Th6, False), (Th7, True)}` $\longrightarrow$ `V4`
- `{(Th4, False), (Th5, False), (Th6, False), (Th7, False), (Th8, False), (Th16, False), (Th9, True), (Th17, True)}` $\longrightarrow$ `{V6, V9}`

The first path is simple and ends in a single code version, `V4`, that is semantically equivalent to the original program. The second path is more interesting, `T7` has two child nodes, that are reached when the comparison is false. Here it is clear that two end nodes are reached,

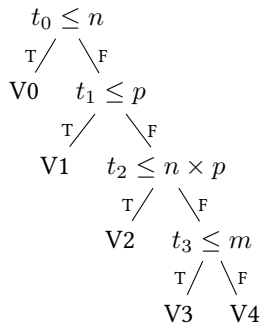


Figure 7: The tree generated by matrix-matrix multiplication. (t_i) is a threshold, (n, p, and m) are the sizes of the matricies show in the code (see 4), and (T) and (F) are indicative of the path we take, based on the threshold comparison.

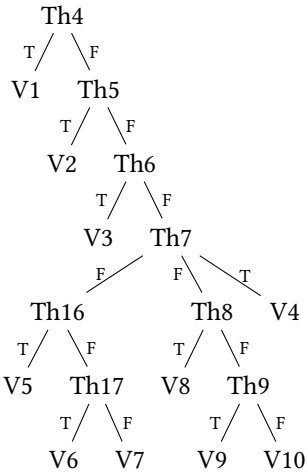


Figure 8: The dependencies between thresholds, of the test program `LocVolCalib.fut`. (Th) is a threshold comparison, (V) is a code version, and (T) and (F) are indicative of the path we take, based on the threshold comparison

namely `(V6, V9)`, and these two code versions are both executed, and together they are semantically equivalent to the original program. This is also important to note, because we could have a forest, instead of a single tree and this would lead to multiple code versions that are all executed, together being semantically equivalent to the original Futhark program.

3 Design & Implementation

Our goal when autotuning a Futhark program is to find the set of values for the threshold parameters that will minimize the running time of the program on the given datasets. There is no way to be completely sure which combination of values will lead to the fastest running time on the given datasets, except to try them all out. This is called an exhaustive search, and it is the way we will find the optimal set of values for the threshold parameters.

As we saw in Section 2 the threshold parameters within a program exhibit a tree structure of dependencies, see Figure 8. Therefore we will start by explaining how we build a structure analogous to these.

3.1 Building the Tree

Every, compiled, Futhark program can be given a flag[2] so the thresholds, and their structure, is printed out. The syntax of the string representing each threshold parameter can be seen in Figure 9. We translate the list of these strings into a tree structure similar to the trees

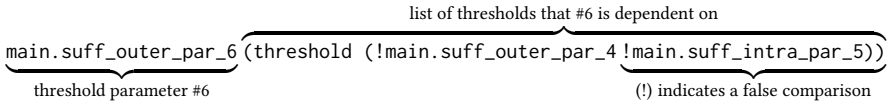


Figure 9: The syntax of thresholds, printed by Futhark.

we have seen in Section 2. We start by finding the thresholds that does not depend on any other thresholds, being true or false, in order for them to be evaluated. These thresholds will be root nodes of their respective trees. In it's current form Futhark will only generate one of these root nodes. In order to make the autotuner adaptable to future changes to the compiler we take into account the possibility that the structure of the thresholds will be a forest. We solve this by creating a root node that has no useful information, called *nullroot*, that will serve as the parent of the forest, this allows us to process a forest the same way we would a tree.

To find the children of one of the root nodes we look for all the thresholds that depend of the root, and make them children of it in the tree. To get their children we look for the thresholds that depend on the node itself and the root that was it's parent. This is done for each layer until there are no more thresholds that can be put as children of a node.

[2] `--print-sizes`. The environment variable `FUTHARK_INCREMENTAL_FLATTENING=1` needs to be set first.

Unlike the theoretical trees, see Section 2, the implemented trees cannot carry information on the edges themselves. This means the information is instead encoded in the children. So each node is a tuple containing the name of the threshold parameter and a boolean denoting the result of the parents threshold comparison that lead to the node. The process of constructing the tree can be seen in Figure 10.

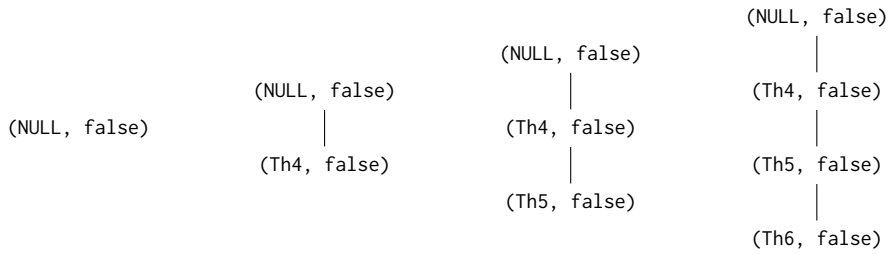


Figure 10: The recusive steps of building the tree structure for the thresholds, with (Th) being a threshold parameter. The list of thresholds for the tree would be `[main.suff_intra_par_5 (threshold (!main.suff_outer_par_4)), main.suff_outer_par_4 (threshold ()), main.suff_outer_par_6 (threshold (!main.suff_outer_par_4 !main.suff_intra_par_5))]`

A possible change in the Futhark compiler that our implementation cannot handle in its current form is if thresholds can depend on multiple direct parents. This change would require a child to carry more information than a single boolean. This problem could be solved by duplicating the problematic subtree, thereby having a version of the subtree that came from a comparisons being true, and one for comparisons being false.

3.2 Limiting Search Space

If we knew nothing about the thresholds, or their structure, we would have to try every configuration of numbers between 0 and whatever the highest number their implementation allows. With a simple program such a matrix-matrix multiplication (which has 4 thresholds, see Figure 7), and assuming an upper limit of 2^{15} which is the threshold default, and therefore a conservative limit, we would end up with $\left(2^{15}\right)^4 \approx 1.153 \times 10^{18}$ configurations. This search would obviously take far longer than any of us would be alive, so it's paramount that we limit the search space.

They way we limit the search space is by using our knowledge of the threshold parameters and only trying values that would result in a different sets of code version being executed on at least one of the datasets. More specifically we will use the knowledge that the comparisons are of the form (`Threshold_value <= comparison_value`). Since the `comparison_value` is constant for a certain dataset there are effectively only two different threshold values that need to be checked for one threshold in isolation. One where it's equal to the `comparison_value` and one where it's `comparison_value` $+ 1$, as these will result in true and false respectively. Of course the thresholds are not in isolation but structured as we've seen Figure 8. So some comparison result in earlier thresholds will make later thresholds in the tree redundant, as they are never compared. We can see this idea in Figure 11. A concrete example would be Figure 7, with the

thresholds $t_i = [10, 20, 30, 40]$, and the matrix sizes $n = 20, p = 30, n \times p = 600, m = 50$. So the predicate of $t_0 \leq n$ would result in taking the `true` branch, however this is also the case for $t_0 = 11$, making these two different configurations have the same dynamic behavior.

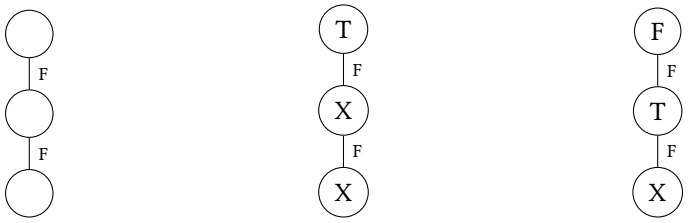
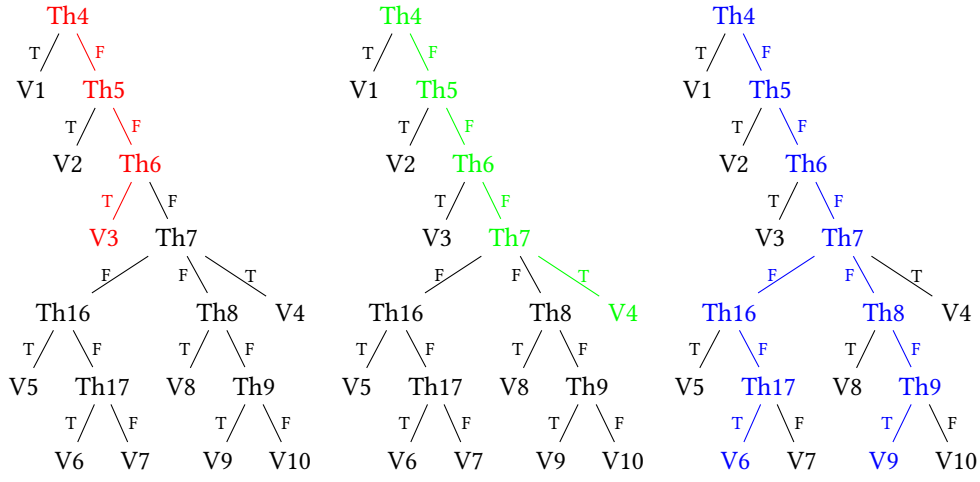


Figure 11: In Figure 11a we see the tree, with *X* denoting that the comparison has not be performed, in Figure 11b we see the case that the first comparison results in true. Since the first comparison resulted in true the next two are never performed, and they can be forgotten. The same idea can be seen in Figure 11c where the first results is false, the second in true, and so the last is never performed.

One thing that is also important is that we are tuning with multiple datasets. Therefore we can not just think of all the unique paths through one tree, instead we need to think of each dataset as having it's own tree with it's own `comparison_value` for each threshold. This means the search space is not every unique path through the tree but every possible unique combination of paths through the many trees we now have. A visual example is show, and explained, in Figure 12.

3.3 Identifying all unique combinations

The values that the thresholds are compared to in each dataset are extracted to a list of values for each threshold. Then they are sorted and duplicates are removed. Since the comparisons are all less than or equal (`Threshold_value <= comparison_value`) comparisons we can simply set the threshold to the smallest `comparison_value` and be sure that the comparison will result in true for all our dataset. It's also possible to construct a value that will guarantee that the comparison will result in false. We do this by taking the largest value for each threshold, where all but the largest dataset will result in false, and then incrementing that value by one. If we then set the threshold to this new value the comparison will be false for the largest dataset as well. We put this false-value at the end of the list of values we extracted from the program. We are left with a list of values for each threshold where the first value will make the comparison for every dataset result in true, the last will make the comparison result in false for every dataset, and the values between will result in true for some and false for others. These lists are then put on their corresponding threshold in the tree we've constructed.
To explain how the paths are found we will use the example tree in Figure 13.



(a) small (b) medium (c) large

Figure 12: We see an example of how three dataset might choose different paths through the tree depending on their size, with the same set of threshold parameters. The general idea is that the longer down we go in the tree the more parallelism is utilized. The three trees here show a unique path through each, and this combination of these three paths, is also unique. We need to check all of such unique combinations. This tree is based on the *LocVolCalb* program. The paths show here are not necessarily the combination of paths any dataset would take based on any combination of threshold values.

The Futhark compiler cannot currently create such a tree, but it highlights some of the changes that might be implemented in the Futhark compiler in the future, and shows that our autotuner will handle such cases.
We construct all possible combinations in a bottom up manner. First all the paths possible in the leaves are found. This is of course trivial as each combination only consists of a single threshold, so the result will just be each value as a singleton list (see Figure 14).

Next we look at the parents of the leaves. Here we need to combine the nodes own values with the children combinations to create every possible combination. There are a two cases that can occur here.
The first case is if we only have a single child for a certain outcome of the comparison, or we have two children that depend on the same results of the comparison.

In Figure 15a we have an example of the first case. Here we take the first value of the node, will always result in true, and combine it with all the combinations we got from the true child. The same is done for the last value of the node, the one that will always have the comparison result in false, and combine it with the combinations from the false child. Then we take the values in the middle and combine them with all the combinations of both children, as these result in the comparison being true or false depending on the dataset. If there was only one child the part that was explained about the other is not performed and

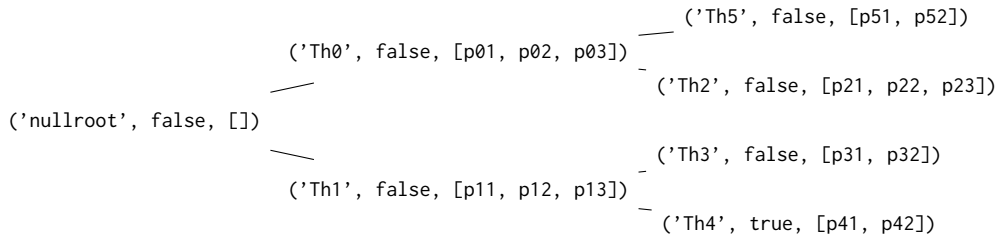


Figure 13: A tree representing the structure of the threshold parameters that is more true to the actual implementation

```
('Th4', true, [p41, p42]) → [[('TH4',p41)],[('TH4',p42)]]
('Th3', false, [p31, p32]) → [[('TH3',p31)],[('TH3',p32)]]
('Th2', false, [p21, p22, p23]) → [[('TH2',p21)],[('TH2',p22)],[('TH2',p23)]]
('Th5', false, [p51, p52]) → [[('TH5',p51)],[('TH5',p52)]]
```

Figure 14: The subtrees that contain only the leafs of the original tree from Figure 13.

instead the true/false only value, whichever type of child did not exist, is put in a singleton list as the result of a comparison with it will lead to a code version.
The example in figure 15a would give us the following result.

```
[[('TH0',p01)],[('TH0',p11),('TH3',p31),('TH4',p41)],
[('TH0',p11),('TH3',p31),('TH4',p42)],[('TH0',p11),('TH3',p32),('TH4',p41)],
[('TH0',p11),('TH3',p32),('TH4',p42)],[('TH0',p12),('TH3',p31),('TH4',p41)],
[('TH0',p12),('TH3',p31),('TH4',p42)],[('TH0',p12),('TH3',p32),('TH4',p41)],
[('TH0',p12),('TH3',p32),('TH4',p42)]]
```

The second case is when there are multiple children that depend on the comparison result, as in 15b. In this case we need to combine the combinations of threshold value of all these children into all combinations of those. These cases occur when the program can end in multiple code versions which means we need to check every combination of those code versions. After combining the results from the children we create every combination of the childrens, now combined, results and the nodes own values.
The example in figure 15b would give us the following result.

```
[[('TH1',p11),('TH2',p21)],[('TH1',p11),('TH2',p22)],[('TH1',p11),('TH2',p23)],
[[('TH1',p12),('TH2',p21)],[('TH1',p12),('TH2',p22)],[('TH1',p12),('TH2',p23)],
[[('TH1',p12),('TH5',p51)],[('TH1',p12),('TH5',p52)],
[[('TH1',p13),('TH5',p51)],[('TH1',p13),('TH5',p52)]]
```

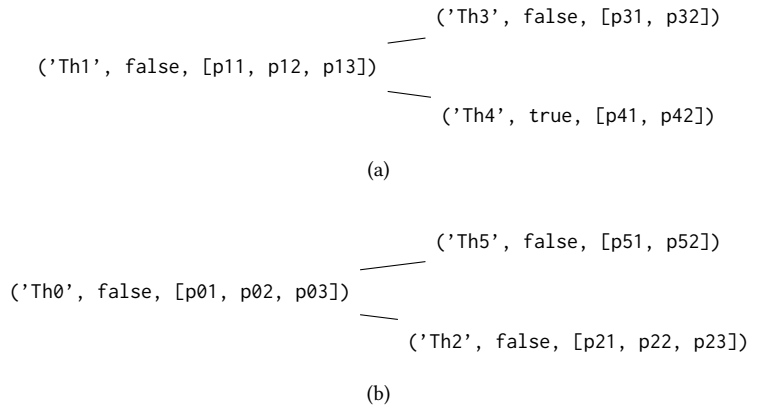


Figure 15: The two subtrees of the root of the tree in figure 13

3.4 Handling Loops

The loop construct that Futhark[3] has will run a piece of code several times with a value that possibly changes each iteration. This means that, unlike the assumption we made earlier, the threshold parameters that correspond to the code inside a loop will be compared multiple times, and that the value is compared to could change in each iteration. The effect of this is that an exhaustive search cannot just assume that each threshold is only compared a single time per dataset. This problem is already trivially solved by the solution put forward in this section, since these extra comparisons will just be added to the list of comparison values and the different combinations they represent will be tried out.

4 Gotta Go Fast!

In this section we will run benchmarks to test our autotunner. There was an existing autotunner for Futhark [12], which was based on OpenTunner [1]. The existing autotunner uses hill-climbing techniques, on a filtered set of possible threshold settings, as opposed to ours, that perform an exhaustive search. We will be comparing the existing autotunner to ours, with executing without tuning being the baseline. We benchmark on four programs [8]. When autotuning we use Futharks build in benchmarking tool `futhark bench`[4]. `futhark bench` defaults to executing each dataset ten times and gives back the average. Along with those ten times, we execute the `bench` five times in creating our graphs, this ends up in fifty runs for each dataset, we do this to avoid deviation in the results.

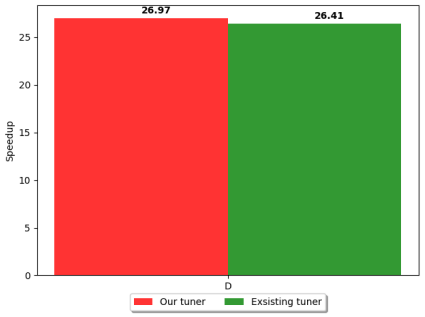
We have benchmarked on a Ubuntu 18.04 system with a **GeForce GTX 1050 4 GB** gpu,

[3]Details regarding loops in Futhark, can be found in the documentation `https://futhark.readthedocs.io/en/latest/language-reference.html?highlight=loop`

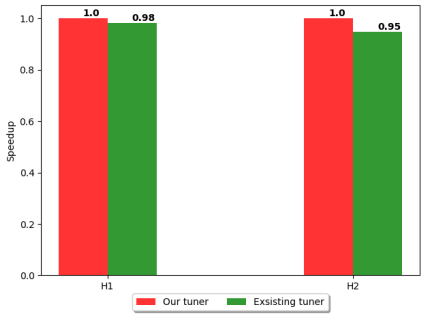
[4]The documentation for `futhark bench` can be found here `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`

and with the NVIDIA driver version: 396.54.

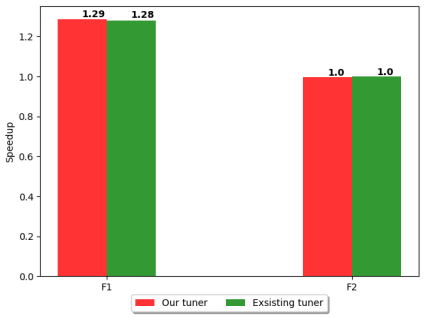
nn, backprop and matmul For the `nn`, `backprop`, and `matmul` program we have two datasets for each `(D)`, `(H1, H2)`, and `(F1, F2)` respectively, with corresponding training sets. In Figure 16a we clearly see that our autotuner produced a set of thresholds that



(a) The result for autotuning the *nn* program, with no tuning being the baseline.



(b) The result for autotuning the *backprop* program, with no tuning being the baseline.



(c) The result for autotuning the *matmul* program, with no tuning being the baseline.

Figure 16: Benchmark for `nn`, `backprod`, and `matmul`

improved `D` massively.

In Figure 16b we see that the neither dataset was affected by our tuning, this would lead us to believe that the default values for the thresholds is a good choice for this program with the datasets `H1, H2`.

And finally in Figure 16c we see that matrix multiplication experience an approximately 30% speedup for `F1` but is equal in `F2`. Matrix multiplication runs well on large threshold parameters, which explain the relatively small speedup, since the thresholds, by default, are quite large.

LocVolCalib There are three datasets for the `LocVolCalib` program, *small, medium* and *large*, as well as a training set for each size.

In Figure 17, we have autotunned for the *small*, *medium* and *Large* datasets, and executed the program on all three. We plot the speedup, or slow down, compared to executing without tuning[5]. In Figure 17 we see that on all datasets the threshold values our auto-tuner

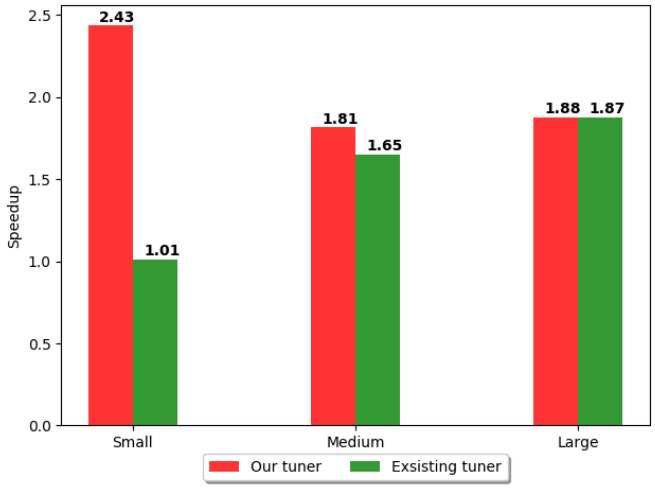


Figure 17: The result for autotuning the *LocVolCalib* program, with no tuning being the baseline.

produced improved the performance of the program. It's also clear that our auto-tuner found a set of values that made the program perform better on *small* and *medium* than the existing auto-tuner. The one problem, which cannot be seen in this graph is the time it takes for each of the auto-tuners to tune the program. For the existing auto-tuner it took around 17 minutes, whereas for our auto-tuner it took a little under 23 hours, which is obviously far to slow to be of use in almost all cases.

Since `LocVolCalib` is so slow when autotuning, we also tested how well it performed when trained on only a subset of datasets, and the result of this can be seen in Figure 18. The motivation behind this experiment was that the number of possible combinations dropped drastically as the number of datasets were reduced. So if near equal results could be gotten with less datasets the tuning time would be more bearable.

[5]Where the default value for thresholds is 2^{15}, which is quite high, meaning that it will favor large datasets

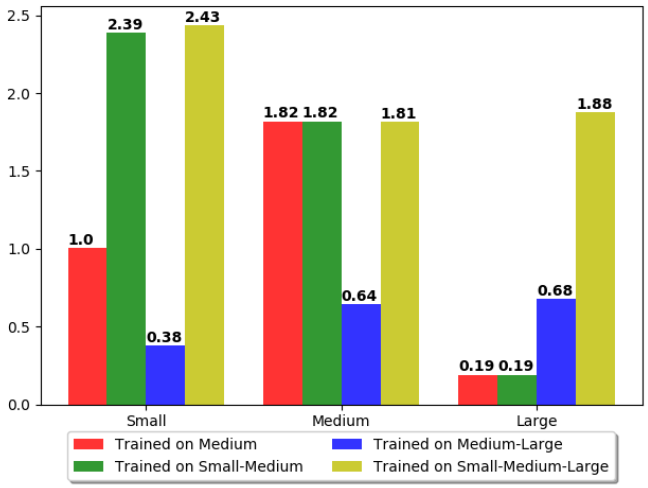


Figure 18: The result for autotuning the *LocVolCalib* program, for different combinations of datasets, with no tuning being the baseline.

5 Reflections & Future work

Our autotuner does what it's supposed to, performs an exhaustive search of all possible combinations of code versions in the given datasets and returns the one that performs the best. This means that in most cases the results it gives are better than the current autotuner. The problem it has is that to perform an exhaustive search and the number of combinations that needs to be checked increases very quickly as you add more datasets. In the `LocVolCalib` program the number of combinations that needs to be checked for one dataset is 13, while the number of combinations that needs to be checked when tune for all three datasets is 5463.

It's clear that our autotuner is fast if the number of datasets are low, so a possible solution is to not use very many. The problem with not using many datasets is that the parameters that are achieved from tuning might not generalize well to new datasets, as their sizes might be totally different. So instead of just leaving out datasets. The possible solution we propose is that it might be possible to create 'synthetic' datasets that, when tuned on, results in threshold values that achieves a good performance on all datasets of interest. This idea can be somewhat seen in our results on `LocVolCalib` (See 18) where training only on the *medium* training dataset resulted in good performances on *medium* and *small* test datasets. This training did not result in a good performance for the *large* test dataset, but this says more about the *medium* training dataset is not a good synthetic dataset, and less that the whole idea wouldn't work.

An oversight we made in our implementation was to not stop an execution, if it was slower than a previous one, so if we had a path e_0 which had a running time of 10 seconds, and

we were actively checking path e_1, then we should stop the execution of e_1 if it exceeds 10 seconds. This is a fairly easy optimization to make and could give a speedup, however the speedup would be dependent on the execution order.

6 Conclusion

Our autotuner works as intended, it finds all the possible combinations of paths for a program, with some amount of datasets, returning the best one for all datasets. The autotuner achieves this with an with an exhaustive search through all these combinations. An exhaustive search guarantees that we will always find the optimal path, which was not the case with the existing autotuner, therefore making our autotuner more reliable.

Being that Futhark is an ongoing research project, changes to it is to be expected. We have taken this into account, and braced the autotuner for these changes. While a Futhark programs thresholds parameters cannot currently be structured as a forest it might do in the future. For this reason we have implemented the autotuner such that it works on forests. While our implementation will not support nodes in the program tree having multiple parents but a method to support this is explained.

With an exhaustive search comes an exponential amount of paths in the worst case, therefore our autotuner gets exponentially slower when tuning for multiple dataset. We saw this in the benchmark for `LocVolCalib` as seen in Figure 17. For that program it took about 23 hours to autotune with all three datasets due to the number of paths being 5463.

We have put forth an idea for future work that can be done on the autotuner, and a optimization to the current implementation. If using a single synthetic datasets, as suggested in section 5, to represent the workload of a program in a single dataset is possible the `LocVolCalib` program will result in 13 paths and therefore a much faster tuning, which could be even faster with the optimization of having slower executions time out during autotuning.

While the GPU tested on is not state-of-the-art, and `LocVolCalib` might be considered large for a Futhark program, we will conclude that the notion of Futhark programs being small enough to comfortably autotune exhaustively, should be reconsidered, unless it is possible to create synthetic datasets that minimizes the amount of datasets needed.

7 User Guide

We have implemented the autotuner as a module to the Futhark program. The modified Futhark program can be found as a submodule in the github repository found at the bottom of the coverpage, as well as here for convenience `https://github.com/SRotendahl/Bachelor.git`. To install it, clone the forked repository, enter the submodule, and follow the instructions from the Futhark documentation `https://futhark.readthedocs.io/en/latest/installation.html`. If you only want the code you can instead just clone the submodule.

The options available in the module can be viewed with the command `futhark autotune -h`. Here is an example of the usage:

```
futhark autotune program.fut --output=program.fut.tuning
        ^^^^^^^^             ^^^^^^^^^^^^^^^^^^^^^^^^^^^
        the module           output file containing the thresholds
```

`--ouput` creates a file containing the thresholds in the format Futhark needs. When executing `futhark bench` it will automatically look for a tuning file, with the same name as the program executed, but with the added `.tuning` extension, we allow for any extension you want to use, since `futhark bench` also allows for this. You can also add the `--tree` flag when executing the autotuner, this will print the tree structure of the thresholds to `stdout` before starting the autotuning. `--ouput` and `--tree` are shortened to `-o` and `-t` respectively. You can also specify the backend that Futhark uses, with the autotuner defaulting to OpenCL.

Here are the specific steps you can perform to try the autotuner:

1. Run `futhark autotune program.fut -o program.fut.tuning`
2. Run `futhark bench program.fut --backend=opencl`. With this you will get the running time of the program using the autotuned parameters, due note that `futhark bench` searches for the `.tuning` file in the current working directory so if it is not in there a path to it should be specified. We specify the backend here due to `futhark bench` defaulting to `futhark-c`.
3. To compare the results without tuning run `futhark bench program.fut --backend=opencl --no-tuning`

References

[1] Jason Ansel et al. "OpenTuner: an extensible framework for program autotuning". In: *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*. 2014, pp. 303–316. DOI: 10.1145/2628071.2628092. URL: https://doi.org/10.1145/2628071.2628092.

[2] Guy E. Blelloch and Gary W. Sabot. "Compiling Collection-oriented Languages Onto Massively Parallel Computers". In: *J. Parallel Distrib. Comput.* 8.2 (Feb. 1990), pp. 119–134. ISSN: 0743-7315. DOI: 10.1016/0743-7315(90)90087-6. URL: http://dx.doi.org/10.1016/0743-7315(90)90087-6.

[3] Guy E. Blelloch et al. "Implementation of a Portable Nested Data-parallel Language". In: *SIGPLAN Not.* 28.7 (July 1993), pp. 102–111. ISSN: 0362-1340. DOI: 10.1145/173284.155343. URL: http://doi.acm.org/10.1145/173284.155343.

[4] Michael Galloy. *CPU vs GPU performance*. June 2019. URL: https://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html. The graph used is an updated version from the article, and can be found here https://github.com/mgalloy/cpu-vs-gpu.

[5] Troels Henriksen. *Streaming Combinators and Extracting Flat Parallelism*. June 2019. URL: https://futhark-lang.org/blog/2017-06-25-futhark-at-pldi.html.

[6] Troels Henriksen. *Why Futhark?* May 2019. URL: https://futhark-lang.org/.

[7] Troels Henriksen, Martin Elsman, and Niels G. W. Serup. "Data-Parallel Flattening by Expansion". In: (2019). URL: https://futhark-lang.org/publications/array19.pdf.

[8] Troels Henriksen et al. *Experimental infrastructure for the paper "Incremental Flattening for Nested Data Parallelism" at PPOPP'19*. May 2019. URL: https://github.com/diku-dk/futhark-ppopp19.

[9] Troels Henriksen et al. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 556–571. ISSN: 0362-1340. DOI: 10.1145/3140587.3062354. URL: http://doi.acm.org/10.1145/3140587.3062354.

[10] Troels Henriksen et al. "Incremental Flattening for Nested Data Parallelism". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: ACM, 2019, pp. 53–67. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295707. URL: http://doi.acm.org/10.1145/3293883.3295707.

[11] nVidia. "NVIDIA CUDA Compute Unified Device Architecture, Programming Guide". In: (2008). URL: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.

[12] Frederik Thorøe. "Auto-tuning of threshold-parameters in Futhark". In: (June 2018). URL: https://futhark-lang.org/student-projects/mette-kowalski-bsc-thesis.pdf.

A Code examples

A.1 Matrix Multiplication - CUDA for GPUs [11, p. 71-73]

Matrix multiplication in CUDA. It is important to note that this will fail for matrices, which shape is not divisible with the block size. We have kept it like this, since this is what NVIDIA puts forth in their programming guide for CUDA.

```
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;

    float Csub = 0;
    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep,
        b += bStep) {

        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];
    __syncthreads();
    }

    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```