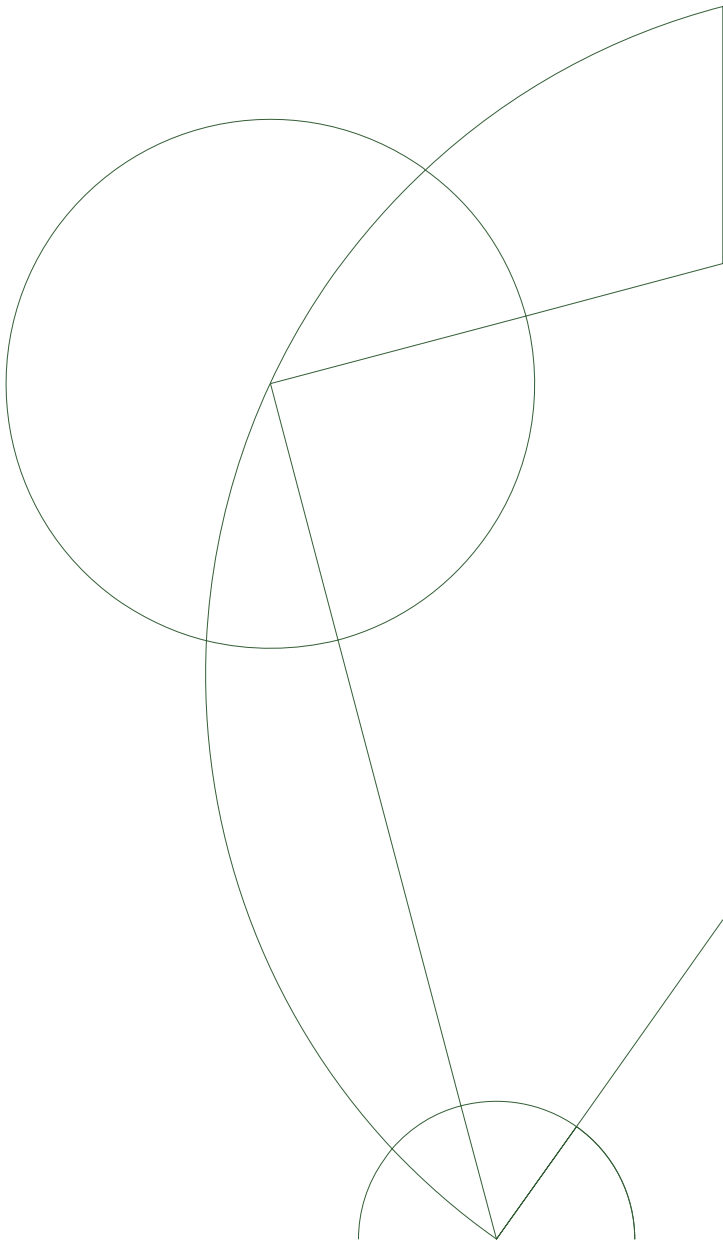
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Frederik Thorøe (vbj532)

Auto-tuning of threshold-parameters in Futhark



Supervisor: Troels Henriksen

June 2018

Contents

Abstract

In the following report a domain-specific auto-tuner for the Futhark programming language is described and implemented. As the current Futhark auto-tuner is an ill fit for tuning Futhark programs, thorough analyses are made to determine how the performance of the auto-tuner can be improved. Two main ideas, both relying on domain-specific knowledge, are introduced to accomplish this. These ideas leads to changes in the Futhark-compiler, which in turn makes the development of an improved tuner possible.

The improved tuner is evaluated using three different programs. It is shown to be significantly faster at tuning these programs than the older tuner, with speedups ranging from 1.9 to 25.5. In addition to this it is shown that the improved tuner produces just as good, or better, results than the old tuner.

1. Introduction

Futhark [6] is a purely functional language which supports nested data-parallelism as well as in-place array updates. The language is an ongoing research project at the Department of Computer Science at the University of Copenhagen. The aim of the language is to shift the main part of the burden of producing efficient GPU-code from the programmer to a heavily-optimising compiler [6].

The field of general-purpose computing on graphics processing units (GPGPU) has been popularized in recent years. The growth in the frequency of CPU-cores began to stagnate in the early 21^{th} century [12], as issues of increased power consumption, growing thermal challenges and semiconductor scaling limits began to become apparent [4]. This led to an increased focus on parallel programming. As GPUs are an example of massively parallel hardware, the interest in using these for general computation, and not just graphics, has followed from this.

Programming GPUs, however, remain a tedious and somewhat daunting task, as it is, to a large extent, done in languages such as OpenCL or CUDA. For both of these languages writing efficient code requires a broad knowledge and deep understanding of both compiler- and hardware-architecture, which makes GPU-programming inaccessible to most programmers. Even when programs are written by specialists with the required knowledge they might, especially for very performance optimized programs, result in non-modular or very hardware-specific, and thus non-portable, code. It is some of these issues, that Futhark tries to tackle [6].

Futhark's solution to the problems mentioned above is to use a heavily-optimising compiler to compile Futhark-programs to OpenCL-code. Futhark features syntax and programming concepts, that most programmers, or at least those with experience with functional programming, will find familiar. By letting the source program be written in Futhark and outsourcing the chore of generating OpenCL-code to the compiler Futhark can there-

fore make the programming of GPUs accessible to a much larger group of programmers. Furthermore the shift from writing hardware-specific code to compiling to hardware-specific code makes it possible to support new architectures merely by writing a new compiler.

This report concerns itself with auto-tuning in Futhark. Auto-tuning is the process of experimentally optimizing the performance of a program by adjusting different parameters at run- or compile time. First the background of the problem is introduced in section 2 along with a motivation for why auto-tuning is relevant and a theoretical look at the problem at hand. In section 3 the process of designing and implementing a new auto-tuner is described. This includes analysis of selected tuning problems, descriptions of modifications made to the Futhark-compiler and breakdowns of the approaches taken to introduce domain-specific knowledge to the auto-tuner. In section 4 the new auto-tuner is evaluated in order to justify the claim, that it is an improvement compared to the current auto-tuner. Finally some reflections about the project and possible, future improvements to the implemented auto-tuner is presented in section 5.

2. Background

2.1 Code versioning in Futhark

Futhark relies on a wide range of optimizations to produce code that runs efficiently on GPUs. While these optimizations has, until recently, relied solely on static analysis performed at compile time, current improvements to the compiler is aimed at evolving a hybrid analysis incorporating both static and dynamic analysis [6].

The reason for introducing dynamic analysis as a supplement to static analysis is perhaps best explained through a simple example. For this consider the following Futhark-program, performing matrix-matrix multiplication of a $n \times m$ and a $m \times n$ matrix.

```
let matmult [n] [m] [p] (x: [n][m]i32) (y: [m][p]i32): [n][p]i32 =
  map (\xr->
         map (\yc->
                reduce (+) 0 (map2 (*) xr yc)) (transpose y)) x
```

Figure 1: Matrix multiplication.

The dynamic analysis is introduced to determine how many levels of parallelism should be exploited at any given run of a program. In the matrix multiplication example in Figure 1 three levels of parallelism are present. However, since the innermost level contains a `reduce` operation this is relatively expensive to exploit. Because of this it can, if the two outer `map`s contains enough parallelism to saturate the GPU, be beneficial to execute the innermost level sequentially and only exploit the two outermost levels of

parallelism. This way the overhead of the innermost `map-reduce` is avoided. Just as introducing unnecessary parallelism can hurt performance, so can not introducing enough parallelism. Thus, if the two outer levels do not provide enough parallelism, the innermost level should also be parallelised.

Whether the two outermost `map`s provide enough parallelism will depend on size and dimensions of the matrices that is multiplied. Since the size of the matrices will not be known at compile time, it is obvious that this problem must be handled with dynamic analysis.

Futhark's solution to the problem is to generate several different, but semantically equivalent, versions of the same code, each exploiting a different amount of parallelism. At runtime the version of the code that exploits the fewest levels of parallelism, while still saturating the GPU on which the program is executed, can then be chosen. This process is dubbed as *incremental flattening* by the Futhark language designers.

To fully understand how incremental flattening influences the execution of a Futhark program, a brief introduction to the GPU execution and memory model is needed. As Futhark compiles to OpenCL the OpenCL-terminology is used. When a kernel, which is the GPGPU-term for functions, is executed, it launches a number of work-items, all of which executes the kernel on different parts of the data. Work-items can be thought of as threads in the OpenCL world. These work-items are organized into a number of work-groups. Work-items share a local memory with all the other work-items in the same workgroup, but cannot communicate with work-items in other workgroups [7]. These constraints much be taken into account when Futhark chooses which version of the code to execute.

When the matrix multiplication program from Figure 1 is compiled with incremental flattening the Futhark-compiler will identify the three levels of nested parallelism. The compiler will then generate three semantically equal versions of the program:

1. One version where each pass of the outer `map`s are assigned to a work-item and the innermost `map-reduce`s are executed sequentially. This versions launches $N \times N$ work-items.
2. One version where the innermost `map-reduce`s are executed in a single OpenCL-workgroup. As the hardware on which a program is executed limits the size of workgroups, this version can only be executed when all the work-items of the `map-reduce` actually fit in a workgroup. This version launches $N \times N \times M$ work-items.
3. One version that relies on Futhark's regular segmented reduction [8] to fully parallelise all three levels of `map`s and `reduce`s. This version also launches $N \times N \times M$ work-items.

Of the three versions above version 1 and version 3 are the most conceptually important, as version 2 is essentially a special case of the more general version 3.

At runtime Futhark will choose which version to execute based on *threshold-parameters*. These parameters will be compared to some data-dependent

values, representing a certain amount of possible parallelism and possibly enforcing some hardware specific constraints. In the rest of this report these comparisons are called *thresholds-comparisons* - or, in some cases, simply comparisons. In our matrix-matrix multiplication example this means that Futhark will always perform the following threshold-comparisons at runtime in order to determine which version of the code should be executed:

```
if t_outer ≤ N × N:
  execute version 1
else:
  if t_intra ≤ M × N × N:
    execute version 2
  else:
    execute version 3
```

In the example above `t_outer` and `t_intra` are threshold-parameters. At a high conceptual level the comparison $t\_outer \leq N \times N$ means that if $N \times N$ provides enough parallelism then execute version 1. All in all we see that the matrix multiplication program has three different execution-paths.

It is worth noting that in reality the comparisons are not quite as simple as they are shown here, as some hardware-specific constraints are imposed. These constraints serve as safety checks when generating the code equivalent to version 2 from the matrix multiplication example. For simplicity these have been omitted here. Due to these hardware-specific constraints version 2 will very rarely be executed when performing matrix-matrix multiplication, as the innermost `map-reduce` will seldom fit into a single OpenCL-workgroup.

To illustrate why choosing fitting values for each threshold-parameter (denoted as *threshold-values* throughout this report) are important consider the benchmark in Figure 2.

This benchmark shows the multiplication of a $2^n \times 2^m$ matrix with a $2^m \times 2^n$ matrix with n varying for run of the benchmark. In order to keep the workload of the different runs of the benchmark constant, while varying the amount of outer parallelism, m is defined as $25 - 2n$. As shown in Figure 2 n takes values in the range $[0 : 10]$. The benchmark is run on the same system as the rest of the benchmarks in this report. For more information see section 3.1.

As it is clear from the graph in Figure 2, version 3 of the generated Futhark-code significantly outperforms version 1 as long as $n < 4$. As the amount of outer parallelism becomes large enough to saturate the GPU with only the two outermost `map`s, which happens at the point where n reaches 4, version 1 starts to outperform version 3. The dip in runtime for version 3 when n lies in the range $[6 : 8]$ is due to the fact that Futharks segmented reduction internally does versioning in a ad-hoc way [8]. It is evident that if one was to execute the same version of the code regardless of the input-data the performance would suffer. By default all thresholds-values are 2^{15}, but these values will not be optimal for all programs or all different hardware. This is obviously the case for the matrix multiplication example on the

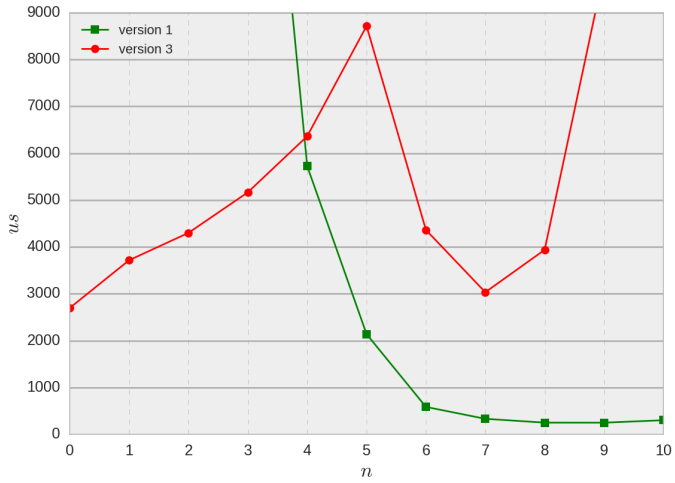


Figure 2: Benchmark of matrix multiplication.

benchmarking system, as the default thresholds would mean that version 1 would only be executed when n reaches 8 making the comparison $2^{15} \leq 2^{2n}$ true.

Unfortunately the current literature provides no examples of suitable analytical models for performance for GPUs. As the performance of a GPGPU-program is dependent on a wide range of factors, such as local hardware, program-behaviour and the dataset it is executed on, it is not feasible for Futhark to automatically choose the optimal code version when a program is being executed. A solution to this problem is to *auto-tune* the threshold-parameters in an attempt to choose threshold-values that ensure optimal, or near optimal, performance.

That no suitable analytical performance model for GPUs has been developed is supported by the fact that auto-tuning is often used to find values for CUDA or OpenCL parameters - a problem that should be easier to solve analytically than Futhark's task of choosing the right code version to execute.

2.2 Auto-tuning in general

Auto-tuning is the process of automatically and experimentally determining the value(s) of one or several parameters with the aim of improving a program's performance [13]. In a GPGPU context these will, as mentioned above, often be CUDA or OpenCL-parameters such as the number of work-items or the size of workgroups. Another example of a problem that exhibits itself well to auto-tuning is compiler flags [2]. Both OpenCL-parameters and compiler flags can have a significant influence on the performance of a given program, and thus choosing the right parameter-values is crucial, if one

wants to achieve optimal, or near-optimal, performance.

Traditionally the values of such performance-influencing-parameters were chosen manually by a specialised programmer, but auto-tuning provides several advantages compared to this approach [11]. First of all auto-tuners can easily search through a much larger search space than what is feasible to do manually. Secondly, as such parameters are often hardware-specific, auto-tuning allows for better portability between different hardware, since the auto-tuner can simply be run on new hardware [2]. In recent years the use of auto-tuning has become ever more widespread [11]. As a result of this a wide range of auto-tuning software has been developed. These auto-tuners generally falls into two categories [10]:

1. Domain specific auto-tuners developed to auto-tune specific problems or languages. Examples of this kind of tuner are ATLAS [14] and FFTW [5].
2. Application-independent auto-tuners, which are purposely general in their approach to tuning and can thus be used for a wide range of problems. ActiveHarmony [11], AtunerRT [10] and CLTuner [9] are all examples of application-independent tuners.

Already existing domain specific auto-tuners will obviously not be relevant for doing auto-tuning in Futhark, and they are therefore not considered any further in this report.

In broad terms the process of auto-tuning typically involves the following three steps [2]:

1. Generating the *search space*, consisting of is the possible tuning parameters and their possible values.
2. Implementing a *cost function* such as execution time, program size or other such metrics.
3. Exploring the *search space* by trying different configurations aiming to minimize the supplied *cost function*. A configuration is a set of tuning parameters and values.

As auto-tuners generally concern themselves with problems where the search space is large, many of them, such as ActiveHarmony [11], generally use different kinds of hill-climbing techniques in order to explore the search space.

For these techniques to prove fruitful each change in a configuration should ideally result in some kind of change in the execution, and thus prompt a measurable change in the result of the cost function. If this is not the case, these hill-climbing techniques have no way to evaluate, whether the change brought it closer to an optimal solution. As analyses in section 3 will show, the tuning of Futhark threshold-parameters exhibit this behaviour. This makes most available application-independent auto-tuners unsuitable for the tuning of Futhark's threshold-parameters.

2.3 Theoretical aspects of the problem

Before moving on to the practical implementation of a new auto-tuner in section 3, the following section introduces the theoretical aspects of the problem.

Letting P be a set of all possible configurations of a given program, and $(t_1 \dots t_n)$ be a specific configuration of n threshold-values we have:

$$P = \{(t_1, t_2, \dots, t_n) \in \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}\}$$

Denoting our cost function as $f(x)$ we wish to have an auto-tuner that solves the following problem:

$$\underset{c \in P}{\operatorname{argmin}} f(c)$$

Solving this problem efficiently is hard, as the search space, P, is very large. Throughout section 3 several different auto-tuners are introduced and described. Each of these try to optimize the tuning process by imposing restrictions on the domain of the threshold-values, and thus reducing the search space. While the practical implementation of the tuners will not be described here, the theoretical aspects of these restrictions will be introduced.

The approach taken by the already existing auto-tuner, which will be discussed in section 3.2, is merely to specify an upper bound of 1,000,000 on each threshold-value:

$$P = \{(t_1, t_2, \dots, t_n) \in \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N} \mid t_i < 1000000\}$$

While this does restrict the domain to some degree, it still leaves a very large search-space for the auto-tuner to explore. Therefore further restrictions are imposed in the comparison based tuner, which will be discussed in section 3.4. For now it is sufficient to notice that this definition further restricts the domain by defining both a upper and lower bound on each threshold-values in a configuration:

$$P = \{(t_1, t_2, \dots, t_n) \in \mathbb{N}\times\mathbb{N}\times\cdots\times\mathbb{N} \mid (\exists (a_i, b_i))[(a_i, b_i) \in \mathbb{N}\times\mathbb{N} \wedge a_i < t_i < b_i]\}$$

Note that these bounds are *not* necessarily the same for all threshold-values in a given configuration.

Though this once more limits the search-space of the auto-tuner a final restriction is imposed in what will be known as the branch based tuner, to be introduced in section 3.6. For this the notion of T being the set of configurations that have already been tried by the auto-tuner, and $e(x)$ being a function that given a configuration calculates the execution-path of this configuration, must be introduced. The restrictions are then expanded as follows:

$$P = \{(t_1, t_2, \dots, t_n) \in \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N} \mid (\exists (a_i, b_i))[(a_i, b_i) \in \mathbb{N} \times \mathbb{N} \wedge a_i < t_i < b_i] \\ \wedge (\forall c \in T)[e((t_1, t_2, \dots, t_n)) \neq e(c)]\}$$

Again further explanation of this approach is saved for later. What should be noted is that the domain of the threshold-values it is now restricted further by demanding that any new configuration does not share its execution-path with an already tried configuration.

These restrictions constitutes the main thoughts behind the optimizations of the auto-tuner. As shown in the following section these ideas are supplemented by more implementation specific heuristics.

3. Design & implementation

3.1 Benchmarking information

During the following section several different auto-tuners for Futhark will be presented and evaluated. These evaluations are all done on a system with a NVIDIA GeForce GTX 1050 Ti graphics processor. As a result all times and speedups reported throughout this report are local to this system.

All tuners have been evaluated using three different programs from Futhark's benchmark suite:

1. The matrix multiplication program shown in Figure 1.
2. The NN-program from the Rodinia [3] benchmark suite ported to Futhark by the Futhark language developers.
3. The LocVolCalib-program from the Finpar [1] benchmark suite which has also been ported to Futhark by the Futhark language developers.

These three programs vary in complexity, ranging from the very simple matrix multiplication to the complex LocVolCalib-program, which has approximately 200 lines of code. It is therefore reasoned that using these three programs for evaluation will give a pretty reliable picture of tuner performances, even though the sample size is quite small.

3.2 Existing auto-tuner

At the outset of this project there was already implemented a basic auto-tuner[1] (from here on denoted as *the old tuner*) for Futhark based on the OpenTuner-framework [2]. The OpenTuner-framework is a open-source framework written in Python and designed for building domain specific auto-tuners. The framework implements all the infrastructure necessary to perform auto-tuning, such as measuring and storing results, as well as a wide range of hill-climbing, and other, techniques used to explore the search space.

As only the hill-climbing techniques allows for purely automatic tuning, i.e. tuning where the atuo-tuner itself will report when it has found a locally optmial configuration, the old tuner was limited to using these. The OpenTuner-framework also employs so called *meta-techniques*, used for running several different techniques and assigning scores the each of these based

[1] Available at `https://github.com/diku-dk/futhark/blob/6908ce4cdbdec8bd569d459d84db57bca0df9bde/tools/futhark-autotune` and in the supplementary `src.zip`-folder

on their results [2]. This makes it possible to use several different hill-climbing techniques at the same time. Since the use of these techniques will be adjusted based on their results, that is, a technique producing good results will be used more than one producing bad results, this ensures that even if one of the techniques get stuck in a unfruitful part of the search space, the tuner can still move on. Preliminary testing showed that limiting the tuner to using hill-climbing techniques did not hurt the performance of the tuner.

The old tuner, as well as all the other tuners described in this report, uses the utility `futhark-bench` to test configurations. When a program is run by `futhark-bench`, the utility will by default run the program 10 times for each of the specified datasets. A dataset is simply some predefined data that will be used as input when running the program. Currently the selection of datasets is fully manual, and thus the results of the auto-tuner is dependent on choices made by the programmer. The three programs used for evaluation in this report has between 3 and 10 datasets.

As tuning is done based on these datasets, it is important that the datasets are representative. Tuning a program using only small datasets will not provide results that can be expected to improve performance if the program is subsequently run with large input data and vice versa. Thus, the datasets should ideally represent a wide range of different input-data. As the programs evaluated in this report is taken from the Futhark benchmarking suite the datasets are assumed to be representative.

The cost function of the old tuner is simply a summation of the runtime for all of the datasets. The same cost function is used in all of the tuners presented in the report.

The old tuner is capable of auto-tuning both OpenCL parameters as well as Futhark's own threshold-parameters. As the focus of this project is the tuning of the threshold-parameters specific to Futhark, the tuning of OpenCL-parameters will not be discussed in any detail in this report, and this capability will not be preserved in new tuners.

The approach taken to tune threshold-parameters by the old tuner is straightforward. Each threshold-parameter is represented in the tuner as an OpenTuner *LogIntegerParameter*. This means that the hill-climbing techniques of OpenTuner will explore the range of possible values of each threshold-parameter on a logarithmic scale. The range assigned to each parameter is $[0 : 1000000]$. The reason for representing the parameters as *LogIntegerParameters* is the assumption that the parameters will in general need to change a lot before it results in a different code version being executed. It is obvious that hard-coding the value ranges in such a way is not an optimal solution. For some threshold-parameters the predefined range might not be broad enough to ensure that the tuner prompts boolean changes in the results of all threshold-comparisons. For others it might be far to broad a range, making the search space unnecessarily large. As no further customizations is made to adapt the OpenTuner-framework to the specific task of auto-tuning threshold-parameters, the old tuner is practically

a general, application-independent tuner targeted at Futhark.

The old tuner struggles when it comes to finding suitable threshold-parameters. To demonstrate this, the old tuner was used to tune the threshold-parameters of the three programs mentioned in section 3.1. The tuner was limited to run for a maximum of 30 minutes. Though tuning can be done for hours or days in the real world, it was felt that on a reasonable new GPU, such as the one used for all tests in this report, tuning for 30 minutes should provide some reasonable results.

To account for the randomness of the tuners hill-climbing techniques each program was tuned 3 times. As the result of tuning can be very dependent on the initial values chosen by the tuner (as can be seen in Figure 4), running the tuner three times hopefully provides a more representative picture of the tuning capabilities than just running it once for a longer period of time.

The OpenTuner-framework will stop tuning when the hill-climbing techniques estimate that a (locally) optimal configuration is found. For none of the three programs did the tuner in any of its runs finish before the allotted time ran out. Figure 3 shows the average number of configurations tried for each program within the time-frame.

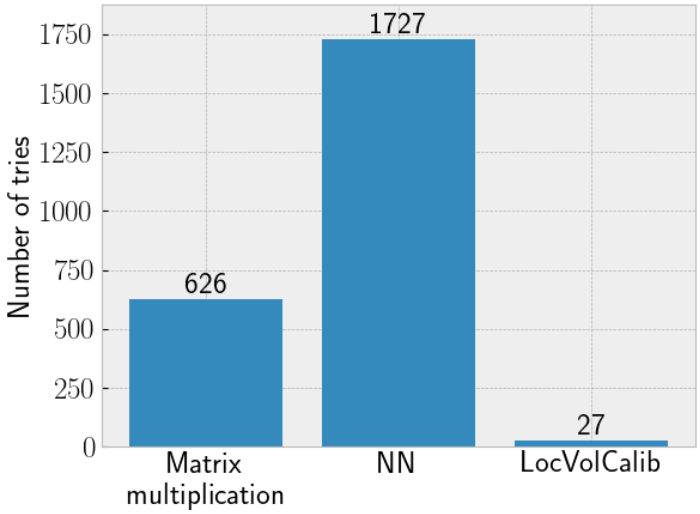


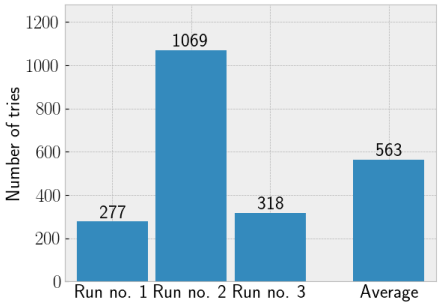
Figure 3: Number of tries managed by the old tuner within 30 minutes - averaged over 3 runs.

As the graphs shows the number of completed tries for each program varies dramatically. For the LocVolCalib-program it takes a long time to try each configuration, and thus only an average of 27 tries is completed. For the two other programs the tuner manages a large amount of tries before its time runs out. However, as noticed above, in none of the cases did the tuner actually estimate that it was done. That the tuner is not able to find

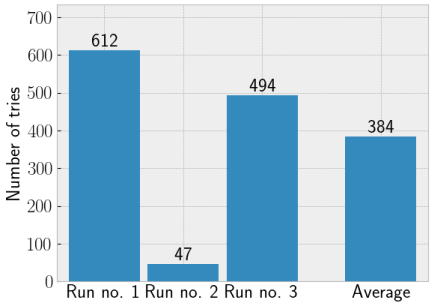
an optimal configuration for the NN-program in over 1700 tries is clearly a problem.

When studying the old tuner it is not only interesting to study how many configurations it tries, but also in how many tries it takes it to find the best configuration for each run. If it consistently finds the best configuration after a relatively small amounts of tries, simply limiting the amount time the tuner is allowed to run for might be a good way to speed up the tuning process. However, as shown in Figure 4, this is not the case.

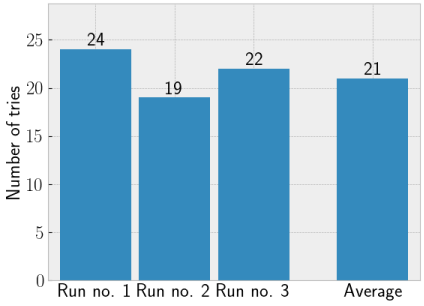
The number of tries to reach the best configuration does not vary significantly for the LocVolCalib-program. This is due to the small number of total tries that the tuner manages to execute with this program. For both of the other programs the number of tries fluctuates to such a degree that simply limiting the number of tries the old tuner is allowed to run does not seem as a feasible solution.



(a) NN



(b) Matrix multiplication



(c) LocVolCalib

Figure 4: Number of tries it takes the old tuner to find the best configuration in a given run.

Note that there is no guarantee that was is above described as the best configuration is in fact the best possible configuration, as the tuner might

discover an even better configuration if it is allowed to run longer.

From these initial analyses it seems clear that to improve the performance of the tuner it must be further adapted to the problem of tuning threshold-parameters.

3.3 Analysis of the threshold-comparisons

In order to investigate why the old tuner performs as badly as it does, when it comes to tuning threshold-parameters, the tuning process of the matrix multiplication program was studied. This program was chosen as its small number of different threshold-comparisons (2) made it possible to enumerate all the threshold-values needed to try all execution-paths of the program.

For this analysis the threshold-comparisons for all datasets were manually extracted from the intermediate code exposed though some of Futhark's debugging options. Using this information each configuration tried by the tuner was tested to see whether it produced a distinct configuration. By this is meant a configuration, which for one or more datasets results in a not yet seen combination of boolean results in the threshold-comparisons. When performing the analysis the hardware-specific constraints, that might influence some of the threshold-comparisons, were ignored. As all possible permutations of the boolean results were enumerated, this should not influence the result in any meaningful way. For results of the tests see Table 1.

	Run 1	**Run 2**	**Run 3**	**Average**
Tries	640	614	625	626
Distinct tries	10	11	13	11
% of distinct tries	1.56	1.79	2.08	1.8

Table 1: Number of tries and distinct tries in the three runs of the tuner.

From this analysis it was clear, that main issue of the old tuner is the problem mentioned earlier on in section 2.2. The range between threshold-values that actually affects a threshold-comparison, i.e. changes the result of a comparison from false to true or the other way around, can be very large. This means that the tuner will try a lot of configurations that does not result in any change in which code-version is executed. As Table 1 shows, only about 1-2% of the configurations tried by the tuner were distinct. Since the tuner has no information about the threshold-comparisons, the only measure it has of each new configuration is its runtime. Naturally the runtime does not change in any significant way when the results of all threshold-comparisons stay the same, and therefore the tuner cannot provide any reasonable information on which its hill-climbing techniques can act. Thus, the tuner is more or less left to guessing which parameters to mutate and whether the value of these parameters should be increased or decreased.

3.4 Comparison based tuner

In an attempt to solve the problem mentioned in the previous section a modification was made to Futhark's compiler by the Futhark language designers. This modification introduced a new flag in each Futhark executable. If this flag is given, the values of the different thresholds-comparisons will be reported by the program.

With this newly available information a new, domain-specific tuner[2] (from here on denoted as *the comparison based tuner*) was developed. Instead of relying on the hill-climbing techniques of OpenTuner, the thoughts behind this tuner was to calculate all the distinct configurations and exhaustively try all of these.

Thus, the main part of the comparison based tuner is the function calculating all the different possible configurations. This is done by extracting all the values for the different threshold-comparisons across all dataset. For each of the comparisons in each dataset, two threshold-values are then generated: one making the comparison false and one making it true. Finally the Cartesian product of all the possible values for all the comparisons is calculated.

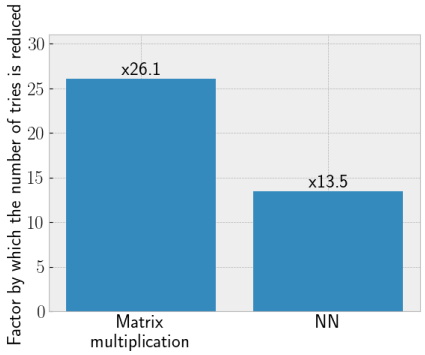
Several steps are taken to minimize the total number of generated configurations. Imagine a threshold-parameter (t) which is compared against two values ($c1$ and $c2$) stemming from two different datasets. We wish to have a configuration where the threshold-value is smaller than $c1$, making the comparison $t \leq c1$ true, as well as a configuration where $t \leq c1$ is false. The same goes for $c2$, where we again want both $t \leq c2 == false$ and $t \leq c2 == true$. By using the middle value of $c1$ and $c2$ we get one threshold-value that satisfy both $t \leq c1 == false$ and $t \leq c2 == true$ at the same time. Eliminating superfluous values is crucial when calculating the Cartesian product. Specifically, if we have m datasets and n threshold-comparisons for each dataset it allows us to reduce the number of configurations from $(2m)^n$ to $(m+1)^n$.

Some threshold-comparisons compares a threshold-parameter against the same value for several, or all, datasets. Expanding on the example from above: If the threshold-comparisons $c1$ and $c2$ satisfies the constraint that $c1 == c2$ we can satisfy both $t \leq c1 == true$ and $t \leq c2 == true$ with the same threshold-value. The same will obviously be the case for $t \leq c1 == false$ and $t \leq c2 == false$. Care is taken to ensure that these types of comparisons does not result in more configurations than necessary being added to the configuration pool, which is the total amount of different configurations that the tuner should try. Thus, for a program with n threshold-comparisons, where m_j denotes the number of different values the jth threshold-comparison takes across all datasets we have a total number of configurations of:

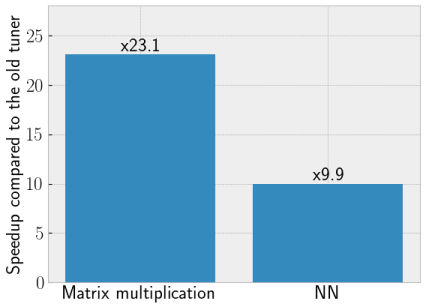
[2] Available in the supplementary `src.zip`-folder.

$$\prod_{j=1}^{n}(m_j + 1)$$

This strategy proved very effective for problems with a relatively low number of threshold-comparisons, such as the NN and matrix multiplication programs. Here, as shown in Figure 5, we see a significant reduction in the number of tries compared to having the old tuner run for 30 minutes. This reduction in tries also leads to a reduction in the time it takes to tune a program on the benchmarking machine, as can be seen also be seen in Figure 5.



(a) Factor by which the number of tries is reduced compared to the old tuner.



(b) Speedup compared to the 30 minutes runtime of the old tuner.

Figure 5: The reduction factor of tries and the speedup of the comparison based tuner compared to the old tuner. Higher is better.

It should be noted, that as this tuner exhaustively searches through every possible configuration one is guaranteed to find an optimal configuration. This is not the case with the old tuner.

Unfortunately, problems arise when attempts are made to tune more complicated problems, such as the LocVolCalib-program. In LocVolCalib 8 threshold-comparisons are made for each of the 3 datasets. 7 of the comparisons compares a threshold-parameter against a unique value for each dataset, while the last comparison only compares the threshold-parameter against two different value across the three datasets. This results in a total amount of different configurations of:

$$(3+1)^7 \cdot (2+1) = 49152$$

As each run of the LocVolCalib-program is quite slow, demonstrated by the fact that the old tuner managed to test less than one configuration per minute, it is obvious that exhaustively searching through this large space of configurations is not feasible. Hence it is clear that further analysis of

the tuning problem is needed to improve the performance of more complex problems.

3.5 Analysis of the dependency amongst threshold-comparisons

While the comparison based tuner, in contrast to the old tuner, has information about the threshold-comparisons it does not have any information about the dependencies amongst these comparisons. In order to illustrate the effect of these dependencies the branching tree of the LocVolCalib-program has been drawn and is shown in Figure 6.

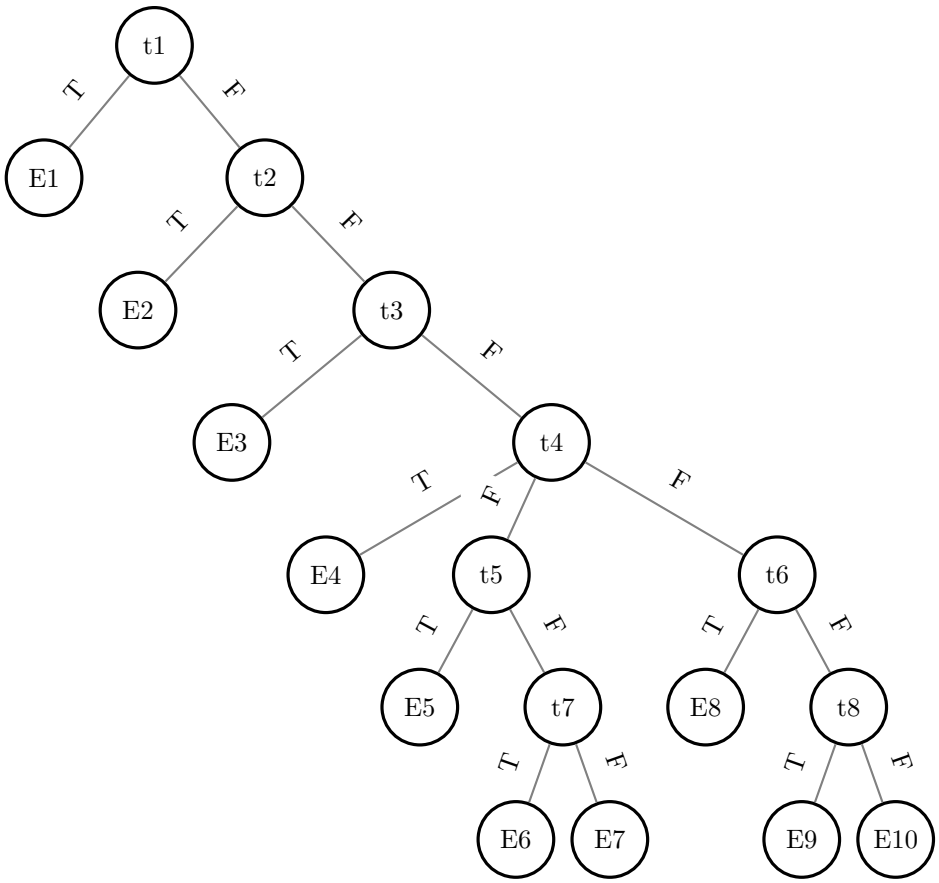


Figure 6: Branching tree showing the dependencies between the threshold-comparisons $t_1 \ldots t_8$ of LocVolCalib. A T along an edge denotes that this path is taken if the comparison is true, and a F denotes that the path is taken if the comparison is false. An node marked with an E denotes that no further comparisons are performed down this path.

As the branching tree shows, there is a heavy dependency among the threshold-comparisons. If t_1 evaluates to true none of the remaining 7 comparisons are performed. This means that a lot of superfluous configurations that only prompt changes in comparisons that are not actually performed

will be included in the search space. Thus, based on the tree in Figure 6 it is obvious that a large portion of the calculated configurations for the LocVolCalib-program can be discarded. A proof of concept, where the branching information from LocVolCalib was used to filter away configurations ending in the same execution-path, further supported this, as it was possible to reduce the number of configurations from 49152 to 286.

3.6 Branch based tuner

Based on the analysis described above yet another change to the Futhark-compiler was made by the Futhark developers. The compiler already supplied each Futhark-executable with a flag that exposed the name of the different threshold-parameters used in the program. This was extended to also include information about the dependencies among the different thresholds-comparisons.

With this new information the comparison based tuner was extended into what will be denoted as *the branch based tuner*[3]. After calculating all possible configurations the tuner calculates the execution path of each configuration for each dataset. Referring to the tree in Figure 6 an example of such an execution path could be $((E6, E7), (E3), (E1))$, where each tuple represents the execution-path of a dataset. Note, that since some levels of the branching tree has several threshold-comparisons a dataset can result in an execution-path with more than one end-node. Based on the this information the configuration pool is filtered. If a configuration has an execution-path is identical to an already existing configuration, it is deleted from the pool, leaving only one configuration for each unique execution-path.

From looking at the branching tree in Figure 6 it is evident that all dependencies among the threshold-comparisons in the LocVolCalib-program rely on a previous comparison being false. This is currently the case for all Futhark-programs. The implementation of the branch based tuner does not, however, rely on this.

The advantage of the branch based tuner is perhaps best shown through a small example. Imagine a program with one dataset and three threshold-comparisons resulting in the branching tree shown in Figure 7.

For this program the comparison based tuner would produce 8 different configurations. These configurations, and their corresponding execution-path, are shown in Table 2. Here T denotes that the configuration makes the thresholds-comparison t_n true, and F that the configuration makes the threshold-comparison false.

From Table 2 it is obvious that there are some redundant configurations in the configurations pool produced by the comparison based tuner. Since the execution path of configuration 2-4 is identical to that of configuration 1 we can eliminate these. The same is the case for configuration 6, since its execution-path is identical to that of configuration 5. Thus, the size of

[3] Available in the supplementary `src.zip`-folder.

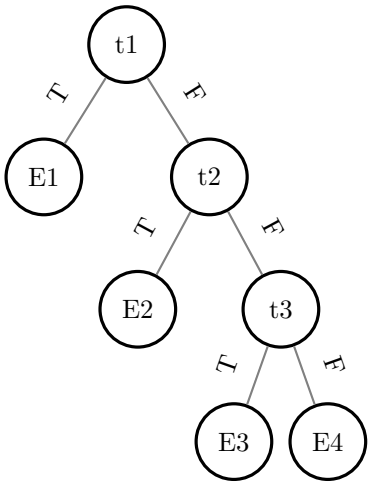


Figure 7: An exemplary branching tree. A T along an edge denotes that this path is taken if the comparison is true, and a F denotes that the path is taken if the comparison is false. An node marked with an E denotes that no further comparisons are performed down this path.

Configurations	***Execution-paths***
1. $t_1 = T$, $t_2 = T$, $t_3 = T$	1. (E1)
2. $t_1 = T$, $t_2 = T$, $t_3 = F$	2. (E1)
3. $t_1 = T$, $t_2 = F$, $t_3 = T$	3. (E1)
4. $t_1 = T$, $t_2 = F$, $t_3 = F$	4. (E1)
5. $t_1 = F$, $t_2 = T$, $t_3 = T$	5. (E2)
6. $t_1 = F$, $t_2 = T$, $t_3 = F$	6. (E2)
7. $t_1 = F$, $t_2 = F$, $t_3 = T$	7. (E3)
8. $t_1 = F$, $t_2 = F$, $t_3 = F$	8. (E4)

Table 2: Overview of configurations and execution-paths for the exemplary program.

the configuration pool is reduced from 8 to 4 configurations. Reducing the number of configurations to try by 4 will only have a very negligible effect on the performance of the tuner. However, as shown in Figure 8, the effect of reducing configurations based on their executing path is much greater for more complex problems.

As expected, this branch based tuner technique proves to be extremely effective with regard to bringing down the number of configurations that the tuner must try. As seen in Figure 8 the improvement is by far most significant in the LocVolCalib-program, which sees a reduction in the number of configurations to try by a staggering factor of 171.9. As this is the most complex program with most dependencies amongst the threshold-comparisons this is to be expected. However the NN-program does also see a factor 4.3 reduction in configurations to try.

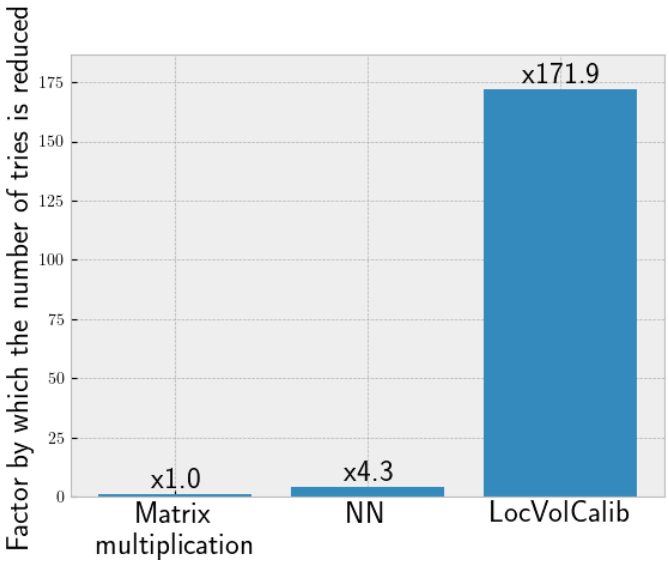


Figure 8: Factor by which the number of configurations to try is reduced with the branch based tuner compared to the comparison based tuner. Higher is better.

The real motivation for implementing the branch based tuner was to reduce the number of configurations for LocVolCalib, as tuning of this program with the comparison based tuner was not feasible. Though the branch based tuner manages to reduce the number of configurations with a factor of 171.9, this still leaves a search space of 286 configurations. As shown in Figure 3 the old tuner was only able to try 27 configurations within its 30 minute time-limit. Thus, it is not feasible to exhaustively search through 286 configurations, if we wish for the tuning to complete within a reasonable time-frame. As a consequence of this, the tuning process of LocVolCalib was further analysed to see if a greater reduction in configurations was possible.

This analysis showed that for the largest dataset certain execution-paths resulted in a 'time-out'. By default `futhark-bench`, the utility used by the tuner to time configurations, reports a time-out if the execution time for a dataset exceeds 60 seconds. Obviously a lot of time-outs will limit the number of configurations that the tuner can try within a given time-frame.

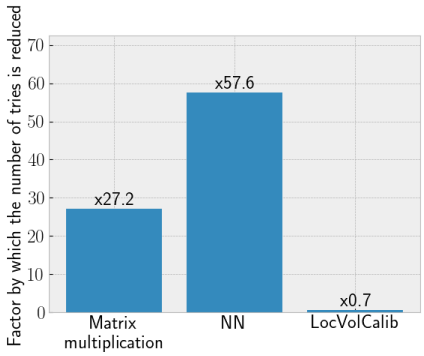
It can safely be assumed that if a dataset times out in a given execution-path once, then all configurations with the same execution path for that particular dataset will time-out. Therefore the branch based tuner was extended to filter away any such configurations. That is, if a configuration with execution-path E_d for dataset d times out, then all other configurations that for dataset d have execution-path E_d will be eliminated.

In addition to this new elimination round a new flag (`--calc-timeout`) was also introduced to the tuner. If this flag is given, the branch based tuner will try to calculate a more fitting time-out value than 60 seconds. This is

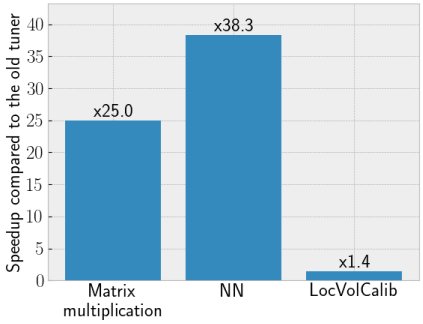
done by first performing a 'vanilla run' of the program where no threshold-values are specified. The time-out value is then set to the runtime of the slowest dataset plus a small safety margin. The reasoning behind this choice is that if a configuration makes a dataset run slower than the slowest dataset of the vanilla run, then it is unlikely that this configuration is optimal. One should note that this flag does pose a theoretical risk of actually eliminating optimal configurations. This can be the case if a configuration causes one dataset to exceed the calculated time-out while speeding the others up dramatically more than any other configuration. Although this behaviour has yet to be encountered, the flag should be used with care. All evaluations of the branch based tuner, and the yet to be introduced final tuner, uses this flag when tuning the LocVolCalib-program.

Tuning the LocVolCalib-program with these new heuristics leads to 11 time-outs. These result in 237 configurations being eliminated. This leaves 38 (plus the 11 that timed-out) configurations to be tried by the tuner. Since we have reduced the time-out limit by calculating a more fitting value for this, trying these configurations is substantially faster than trying the 27 configurations managed by the old tuner. The branch based tuner is therefore able to exhaustively search through this reduced search space in just over 20 minutes on the benchmarking system.

Using the technique described above we can compare the branch based tuner to the old tuner. The results are shown in Figure 9.



(a) Factor by which tries is reduced compared to the old tuner.



(b) Speedup compared to the 30 minutes runtime of the old tuner.

Figure 9: The reduction factor of tries and the speedup of the branch based tuner compared to the old tuner. Higher is better.

The number of tries needed to find a configuration is reduced by a factor of 27.2 for matrix multiplication and a factor of 57.6 for the NN-program. Note that for the LocVolCalib-program the number of tries is increased. This is due to the fact that the comparison based tuner can manage more tries in less time because of the introduced heuristics. All three programs experience a speedup. This speedup is most significant for the simpler program. Matrix

multiplication is 25 times faster to tune with the branch based tuner and the NN-program is 38.3 times faster to tune. For the LocVolCalib-program the speedup is more modest, as the program is only 1.4 times faster to tune, and the real improvement here lies in the fact that more configurations are tried within a shorter amount of time.

Thus, the branch based tuner is capable of tuning all three programs used throughout this report in a somewhat reasonable amount of time. Though the strategy of reducing the search space enough to exhaustively search through it has proven itself with these three programs, this approach might not work for all programs. Some programs might have such a large number of datasets and/or threshold-comparisons, that even all the steps taken to minimize the search space will not be sufficient. To accommodate such problems a fourth, and final, version of a Futhark tuner is introduced.

3.7 Final, combined tuner

The final, combined tuner[4] (from here on denoted at *the final tuner*) is, as the name suggest, a combination of the old tuner and the branch based tuner. This tuner is, as opposed to the comparison and branch based tuners, once again based on the OpenTuner-framework. If the ambition is that a tuner should be able to tune large and complex programs, the idea of performing exhaustive searches through a search space must be discarded. In this case tuning must resort to using hill-climbing techniques. The inclusion of concrete knowledge about execution paths of a Futhark-program evolves the general old tuner into a domain-specific tuner specialized at tuning the threshold-parameters of Futhark.

For such a combined tuner to be effective, steps must be taken to eliminate the problems that made the old tuner ineffective. In broad terms these steps involve two main additions to the old tuner:

1. Using branching information to make sure that configurations with the same execution-path as a previously tried configuration is not tested.
2. Using the values of each threshold-comparison to provide more fitting values for the tuner to try.

Regarding (1) this is simply done by storing the execution-path each time a configuration is tested. Before a new configuration is tried, its execution-path is then calculated, and if this execution-path has already been tested in a previous run, the result of this run is used without actually testing the configuration. Configurations resulting in time-outs are also eliminated in a similar way to the approach taken by the branch based tuner. This greatly reduces the number of configurations that the tuner has to try.

Regarding (2) two different approaches were considered. The fist approach was to, just as in the comparison based tuner, calculate values that for each dataset make all the threshold-comparisons both true and false and

[4] Available at `https://github.com/diku-dk/futhark/blob/67f90e1000ce30f14810748f505bd8e919198d00/tools/futhark-autotune` and in the supplementary `src.zip`-folder.

then simply using the list of these values instead of a value range. The second approach simply sets the range of possible values for each threshold-parameter based on the minimum and maximum values that this parameter is compared against. Since hill-climbing techniques can only be used with the latter approach, and since the motivation for the final tuner was to reintroduce these techniques, this was chosen.

The change from exhaustively searching though a set of configurations to using hill-climbing techniques does come with some trade-offs. First off it might increase the time it takes to tune a program, as the tuner does not know whether it has already covered every possible execution-path. Secondly, since the tuner might find a locally optimal configuration, and thus stop tuning before it has covered every possible execution-path, it removes the guarantee of finding the optimal configuration. These trade-offs are deemed as being worth the capability of tuning extremely complex problems. In section 5.4 a possible solution to these trade-offs is presented, but it has not been implemented.

As will be shown in greater detail in section 4 this tuner is, just as the branch based tuner, capable of tuning both the matrix multiplication, NN and LocVolCalib-program within a reasonable time-frame.

3.8 Technical aspects

Section 3 introduces four different tuners:

1. The old tuner
2. The comparison based tuner
3. The branch based tuner
4. The final, combined tuner

All four tuners are written in Python. While the comparison based and the branch based tuner does not build upon any existing auto-tuning framework, the old and the final tuner are both based on the OpenTuner-framework. As OpenTuner is only compatible with Python 2.x, the final tuner requires Python 2.

As mentioned earlier the final tuner relies on certain information being exposed through different flags that can be passed to a Futhark executable. For information about the different threshold-comparisons the flag `-L` is used. When this flag is passed to a Futhark executable it will, alongside some other information about hardware etc., produce information in the style shown below:

```
Compared suff_outer_par_4793 <= 1024.
Compared suff_intra_par_4833 <= 1048576.
```

Using a regular expression the threshold-comparisons can then easily be extracted. Passing the flag through the `futhark-bench` interface will provide information about the threshold-comparisons for each dataset with which the program is executed. It is worth noting that `-L` will report every possible threshold-comparison, even if the comparison is nested in a branch that is not executed. Thus, it is sufficient to extract this information once

at the beginning of the tuning. Since the final tuner relies on this behaviour, it will have to be modified, if this is changed in later version of Futhark. If they become necessary these modifications should be fairly uncomplicated.

Exposing information about the dependencies of the threshold-parameters is done by passing the `-print-sizes` flag to a Futhark executable. Again information about all dependencies, and not only the ones relevant to the specific run of the program, are exposed. Thus, it is once more sufficient to extract this information once. An example of the information provided when this flag is passed is shown below:

```
suff_outer_par_4793 (threshold (!suff_outer_par_4748 \
!suff_outer_par_4653))
suff_intra_par_4833 (threshold (!suff_outer_par_4793 \
!suff_outer_par_4748 !suff_outer_par_4653))
group_size_4891 (group_size)
num_groups_hint_4893 (num_groups)
num_groups_hint_4970 (num_groups)
group_size_5041 (group_size)
tile_size_6496 (tile_size)
```

As shown the flag also provides information about the different OpenCL-parameters, but as the tuners in this report only concerns themselves with the threshold-parameters this information is not used. The dependency information is encoded in the nested parenthesis of the threshold-parameters. If a dependency is prepended with a exclamation mark, the comparison using this dependency must be false, for the threshold-parameter to be used. If not the comparison must be true. As mentioned earlier Futhark presently only has dependencies that rely on previous comparisons being false, however the tuner is not limited by this implementation detail, and if the behaviour changes in the future no modifications should be necessary.

As the dependency information also includes dependencies that are later eliminated through some optimizations in Futhark, care must be taken to ensure that only relevant dependencies are processed. This is done by only processing dependencies that are also reported as threshold-parameters themselves. In the example from above this means that for `suff_outer_par_4793` both of its dependencies are eliminated, and for `suff_intra_par_4833` only the dependency of `suff_outer_par_4793` preserved.

As stated in section 3.2 and section 3.7 the two tuners based on the OpenTuner-framework has been limited to only using hill-climbing techniques. This has been incorporated into the final tuner, but not the old tuner. For the old tuner the limitation can be imposed by supplying the following flags when tuning:

```
--technique=RegularNelderMead --technique=RandomNelderMead \
--technique=RightNelderMead --technique=MultiNelderMead \
--technique=RandomTorczon --technique=RegularTorczon \
--technique=RightTorczon --technique=MultiTorczon
```

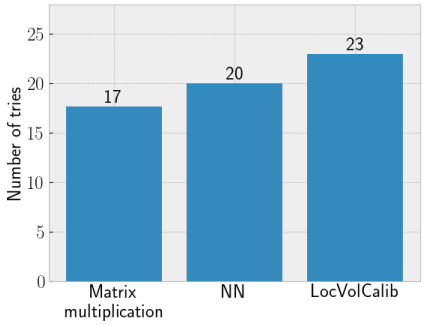
Auto-tuning is often done in parallel on several different systems at a time. Since all available information needed for tuning is currently available

at the outset of this, and since the OpenTuner-framework supports running tests in parallel [2], this approach should in theory also be possible with the final tuner. However, as this has not been tried there might be some details in the interaction with the OpenTuner-framework that prevents this.

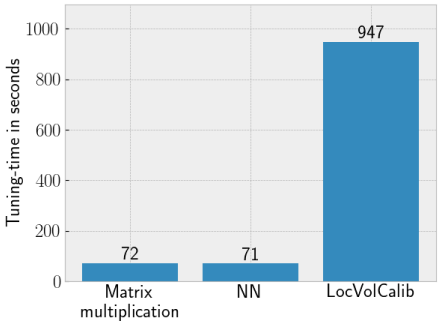
4. Results

4.1 Evaluation of the final tuner

One of the main issues with the old tuner was the amount of time it had to run. Remember from section 3.2 and Figure 3 that for neither the matrix multiplication, the NN or the LocVolCalib-program did the old tuner complete within the 30 minutes time-limit. The final tuner, on the other hand, saw all runs complete within the allotted time.



(a) Number of tries, averaged over 3 runs, before a configuration is reported by the final tuner.



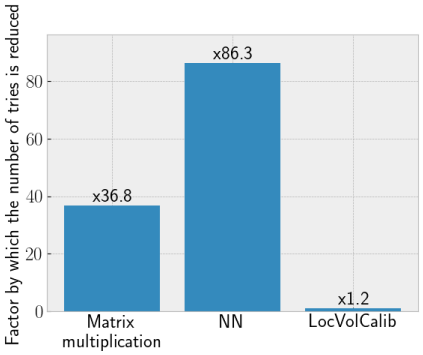
(b) Seconds, averaged over 3 runs, that it takes to complete tuning with the final tuner.

Figure 10: Number of tries and seconds it takes for the final tuner to report a locally optimal configuration.

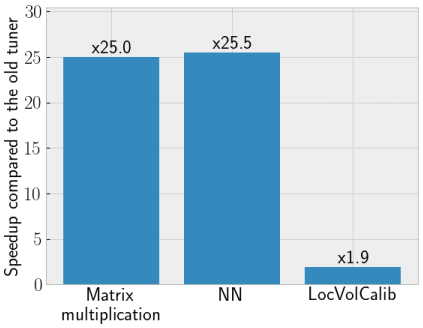
Figure 10 shows the number of tries, averaged over 3 tuning runs, that was completed for each program before a locally optimal configurations was reported by the final tuner, along with the average number of seconds it took to complete tuning. Here a try is a configuration that is actually tested by the tuner, i.e. a configuration with an execution-path that has not yet been tried.

Figure 11 shows the factor by which the number of tries has been reduced by the final tuner as well as the resulting speedup in tuning time. Note that all the speedups reported in this section is with respect to the 30 minute time-limit imposed on the old tuner. Had the old tuner been allowed to run for as long as it would take it to report a locally optimal configuration, these speedups would obviously increase.

For both the matrix multiplication and the NN-program the number of



(a) Factor by which the number of tries has been reduced by the new tuner compared to the old tuner.



(b) Average speedup in the time it takes to find a final configuration for the final tuner compared to the 30 minute timeout by the old tuner.

Figure 11: The reduction factor of tries and the speedup of the final tuner compared to the old tuner. Higher is better.

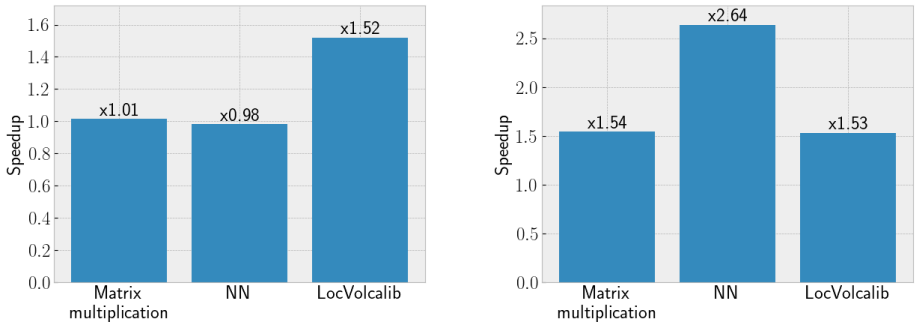
tries has been reduced by a significant factor. For the LocVolCalib-program this reduction is more subtle.

Both the matrix multiplication and NN-program has a speedup factor around 25, which must be categorized as very substantial. For the LocVolCalib-program the speedup is more modest, yet still shows an improvement of almost 100%. Furthermore, it shows that the final tuner was actually able to find a locally optimal configuration and finish tuning in less than the 30 minutes that the old tuner was allowed to run.

Decreasing the number of tries it takes to report a configuration does not do much good if the reported configuration is not a suitable one. Certainly it is also needed to evaluate these and see how the configurations found by the final tuner compares to the ones found by the old. Figure 12 shows this evaluation.

From Figure 12 we can see that for the two simple programs, matrix multiplication and the NN-program, the configurations produced by the final tuner does not result in any speedup, compared to the configurations produced by the old tuner. In the case of the NN-program we actually experience a minor slowdown, but this is due to the fact that runtime will vary slightly across runs, as the execution-path of the configurations are, in fact, identical. For the more complex LocVolCalib-program we see an average speedup of just about 50%. Comparing the runtime of a auto-tuned program to the runtime of the same program without auto-tuning results in a speedup of at least 50% for all three programs.

One should note that the same program, with the same datasets, has been used for both auto-tuning and evaluation. Thus, no cross-validation has been performed, but since the problem of auto-tuning is not a predictive



(a) Speedup compared to configurations found by the old tuner.

(b) Speedup compared to no tuning at all.

Figure 12: Speedup of the configurations found by the final tuner compared to the old tuner and to no tuning at all. Higher is better.

problem, this should not influence the results in any significant way.

Finally it is interesting to return to the matrix multiplication problem from section 2.1 and Figure 1. To see if the tuned version of the matrix multiplication problem actually changes from one code version to another at the right time, the run is repeated with a configuration produced by the final tuner. As seen in Figure 13 the tuned program does in fact change from version 3 to version 1 just as n becomes large enough.

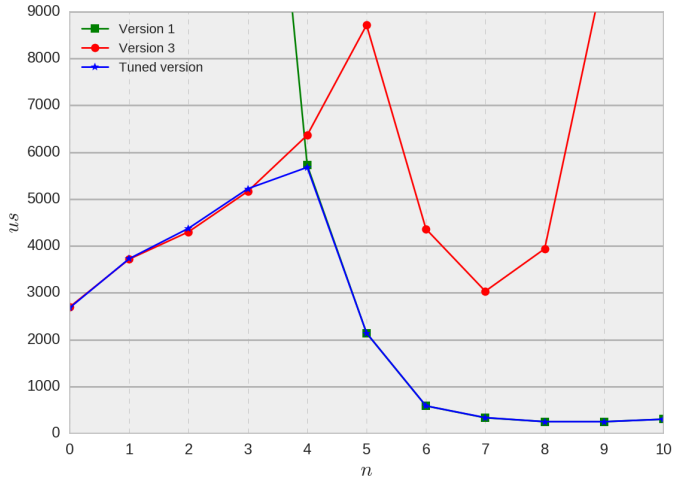


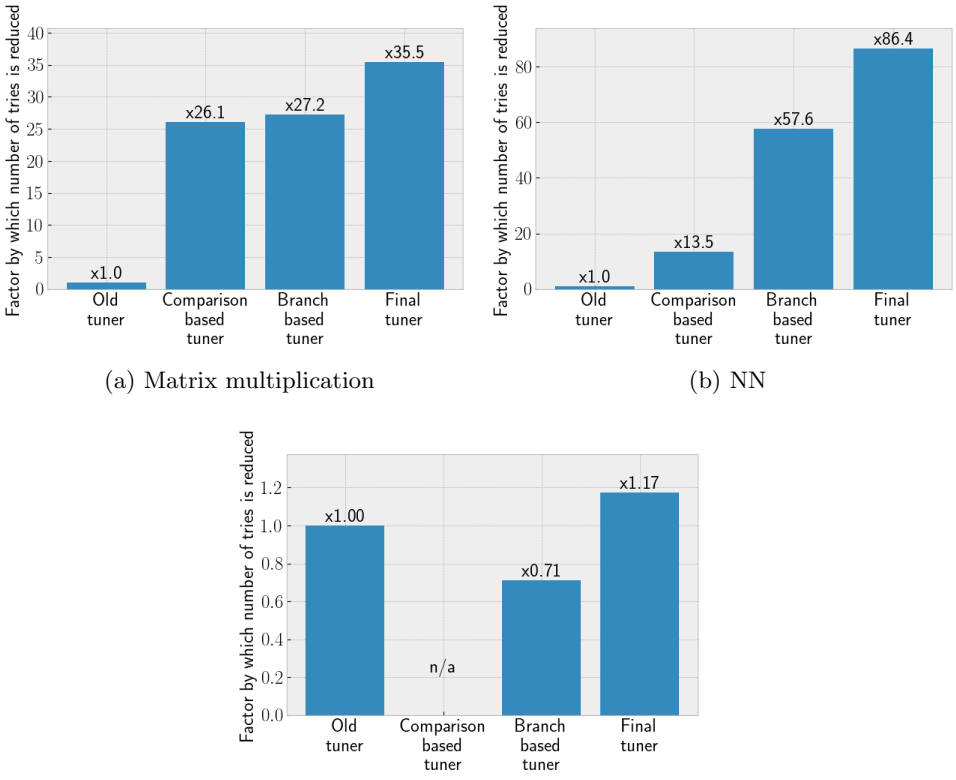
Figure 13: Execution time of matrix matrix multiplication as a function of n for the two code-versions and the tuned program.

All in all it is shown that the final tuner is functional, and that it is an

improvement over the old tuner. It produces configurations that are faster than running a program that is not tuned. These configurations perform at least as well as the configurations produced by the old tuner. In the case of the LocVolCalib-program we even see a speedup of 50%. Furthermore the final tuner is significantly more effective than the old tuner. It tries far fewer different configurations and thus speeds up the tuning process with a factor between 1.9 and 25.5.

4.2 Overview of all tuners

Besides the old and the final tuner two other tuner have been presented in this report. Figure 14 and Figure 15 presents an overview of how all tuners perform compared to the old tuner.



(a) Matrix multiplication

(b) NN

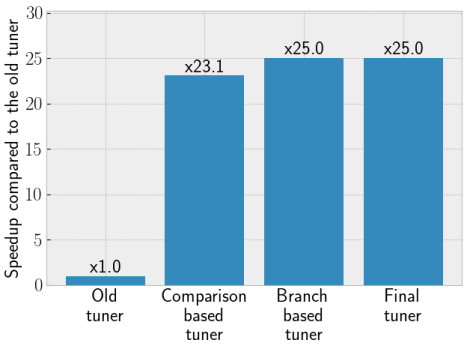
(c) LocVolCalib

Figure 14: Factor by which the number of tries to find a locally optimal configuration is reduced compared to the old tuner for all three programs. Higher is better.

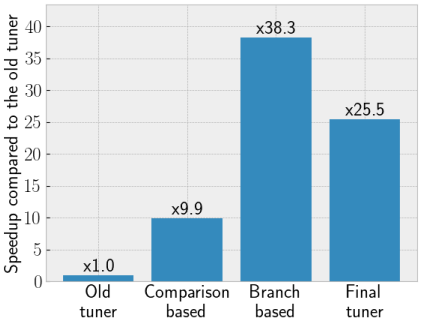
Figure 14 shows the factor by which the number of tries it takes each tuner to find a locally optimal configurations is reduced compared to the number of tries of old tuner. Again remember that the old tuner did not report a locally

optimal configuration within the allotted 30 minutes. It is interesting to note how the heuristics introduced in the branch based tuner only had a minimal effect for the very simple problem of matrix multiplication. For the more complex NN-program the effect was much more substantial. Regarding the LocVolCalib-program we see that the number of tries is actually increased for the branch based tuner, since it is able to try configurations more efficiently. As the comparison based tuner would have to try over 49,000 configurations when tuning the LocVolCalib-program this was never relevant. Therefore no numbers are reported for the LocVolCalib-program and this tuner in neither Figure 14 or Figure 15.

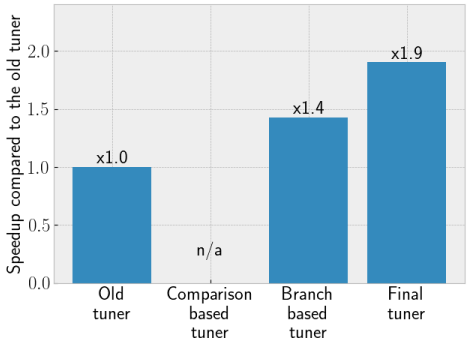
In Figure 14 the effect of discarding the exhaustive search in the final tuner is also evident. Since the final tuner, instead of naively searching exhausting the search space, stops when it has found what it estimates to be a locally optimal configuration it is able to reduce the number of tries compared to the comparison and branch based tuners.



(a) Matrix multiplication



(b) NN



(c) LocVolCalib

Figure 15: Speedup in the time it takes to find a final configuration compared to the 30 minute timeout by the old tuner. Shown for all tuners and programs. Higher is better.

Figure 15 shows the speedup of the tuning process for each program rel-

ative to the old tuner. For the two simpler programs, matrix multiplication and NN, the branch based tuner is actually just as fast, or faster, than the final tuner. This is the case even though the latter tries less configurations, and is the result of one of the trade-offs described at the end of section 3.7. While the branch based tuner will stop tuning as soon as it has exhausted the search space, the final tuner spends time looking for new configurations until it estimates that a locally optimal configuration has been found.

That the tuning of the LocVolCalib-program is faster when using the final tuner than when using the branch based tuner is a result of the final tuner finding a locally optimal configuration before trying all possible configurations. This seems to indicate that while the exhaustive search of the branch based tuner is faster for simple programs, the final tuner is better at solving more complex tuning problems.

5. Reflections and future work

5.1 Cost function

When auto-tuning, the choice of cost function influences the result. As mentioned in section 3.2 the cost function used by all the tuners described in this report is the total runtime. This choice will heavily favor configurations that improve the longest running datasets. While this could be remedied by using an alternative cost function, such as the geometric mean of runtimes, it is reasoned that current cost function is a reasonable choice. Since the benefit of improving the runtime of long running datasets is obviously bigger than improving the runtime of datasets that finish faster, it seems appropriate to use a cost function that leads to just this behaviour.

5.2 The arbitrariness of threshold-values

The current implementation of the tuner is evaluating different configurations against a finite number of datasets. Thus, when the tuner reports a final configuration it might be optimal for the datasets used when tuning, but the reported threshold-values will still be arbitrary and dependent upon the datasets.

An straightforward solution to this problem is to introduce more datasets. This solution does, however, come with a trade-off, as it will slow down the tuning process. In addition to this, blindly adding more datasets will not necessarily solve the problem. Imagine a simply case of a program with three datasets and one threshold-comparison. Across all three datasets this program compares threshold-values against c_1, c_2 and c_3 where $c_1 < c_2 < c_3$. If the tuner reports the optimal threshold-value to lie between c_2 and c_3, adding a fourth dataset where $c_4 < c_1$ or $c_3 < c_4$, will not affect the threshold-value chosen by the tuner. Instead c_4 must satisfy $c_2 < c_4 < c_3$ to have any effect on the arbitrariness.

If this problem is to be solved, the tuner must therefore first identify an optimal configuration for the initial set of datasets before new datasets can be added. Generating and adding new datasets on the fly should be possible for some programs such as matrix multiplication. Returning to the matrix multiplication example and the graph in Figure 13 it should be possible for the tuner to automatically generate a new dataset where n lies between 3 and 4. This process could then be repeated a number of times, each time introducing a new dataset where n lies in the range where the shift from version 1 to version 3 happens. Repeating this process enough times will lead to less arbitrary threshold-values.

For other programs, such as the LocVolCalib-program, this auto-generating of datasets might prompt bigger problems, either due to the complexity of the program itself or due to the complexity of the datasets. The strategy outlined above has therefore not been implemented in the final tuner. It could, however, be an interesting approach to examine in the future.

5.3 Limited number of evaluation programs

As mentioned in section 3.1, and shown throughout the report, only three programs have been used to evaluate the performance of the different tuners. This is clearly not a very elaborate evaluation. The three programs do, however, represent quite a broad scope of different types of programs, with matrix multiplication being a very simple program and the LocVolCalib-program being one of the most complex in Futhark's benchmarking suite. Due to this fact, the current evaluation is deemed adequate with regards to concluding whether performance of the new tuner is better than that of the old. However evaluating the tuner against a wider range of programs might certainly be interesting, if not only to further justify the claims made in the report.

5.4 Future work

As mentioned in section 5.2 a future improvement to the tuner could be to automatically generate new datasets in order to reduce the arbitrariness of the threshold-values. Whether this is indeed practical is not clear, as naively generating new datasets that represent a change in workload might be difficult for some programs. As so further investigations and analysis would be needed.

In section 3.2 it is mentioned, that the old auto-tuner was also capable of tuning OpenCL-parameters. As the focus in this report has been the auto-tuning of threshold-parameters this is currently not possible with the final tuner. Extending the final tuner with this functionality should, however, be trivial. This might be considered in the future.

Another possible improvement would be to introduce the possibility of having the tuner perform an exhaustive search, i.e. executing the branch based tuner described in section 3.6 in certain cases. As shown in Figure 15

the branch based tuner can be faster than the final tuner when tuning simple programs. This is due to the fact that the branch based tuner will stop tuning as soon as it has exhausted the search space. The final tuner, on the other hand, has no notion of how many different configurations are contained within the search space. Thus, it must wait for its hill-climbing to estimate that a locally optimal configuration has been found. Furthermore, the branch based tuner guarantees that every possible configuration is tried - a guarantee that cannot be given with the hill-climbing techniques of OpenTuner.

Thus, it might be preferable to use the approach of the branch based tuner in the cases where the search space can realistically be exhaustively explored. This possibility could be implemented by introducing an exhaustive search technique to the OpenTuner-framework. The tuner could then either let the user manually choose to run this technique by passing a special flag to the tuner, or it could alternatively try to estimate the size of the search space and automatically choose an appropriate search technique itself.

6. Related work

There is no shortage of available, application-independent auto-tuners, which could, in theory, be used to tune Futhark's threshold-parameters. However, as the work done in this report has shown, there are a lot of advantages to be gained from applying domain-specific knowledge to the auto-tuning process. Furthermore, some of the available auto-tuners exhibit characteristics, that make them unfit for use with Futhark.

The ActiveHarmony-auto-tuner [11] is an example of just this. The tuner is a compiler-based auto-tuner [11], which means that its use-case is to automatically generate different versions of a program and test these to find the optimal one. As mentioned in section 2.1 this is already implemented in Futhark, and since the choice between code-versions is made at runtime, Futhark does not lend itself to be used with a compiler-based auto-tuner.

AtunerRT [10] and CLTuner [9] are, on the other hand, both online auto-tuners, meaning that they work by manipulating parameters at runtime. Both tuners are, however, targeted at tuning individual OpenCL-kernels and not whole programs. Both tuners work through their own API, which must then be integrated into the program one wishes to tune. While it would certainly be possible to update the Futhark-compiler to include these API-calls in each compiled Futhark-program, these tuners would still suffer from the lack of domain-specific knowledge that this report has shown to be of great importance to an effective tuning.

7. Conclusion

This report introduces a new, domain-specific auto-tuner for tuning Futhark's threshold-parameters. The final tuner is build upon the OpenTuner-framework

for building domain-specific tuners. Through analyses of Futhark-programs several approaches has been developed to increase the performance of the final tuner compared to the already existing auto-tuner. These analyses has also led to modifications to the Futhark-compiler, making these new approaches possible.

Information about the execution paths and threshold-comparisons of a given Futhark-program is extracted during tuning, and this domain-specific knowledge is then applied to minimize the time the tuner spends trying threshold-values that will not provide any performance gain. In addition to this the information is also used to define a more limited and precise search space for the tuner to search through.

The final tuner is evaluated by auto-tuning three selected Futhark programs. These evaluations show that the performance of the configurations found by the new tuner is on par with, or better than, the configurations found by the old tuner. Furthermore for all three programs the final tuner is capable of performing a 'full tuning' - i.e. a tuning where the hill-climbing techniques of the tuner estimates that an locally optimal configuration has been found - in less than 30 minutes on the testing machine. This was not possible with the old tuner. The speedup in the time it takes to tune the programs ranges from 1.9 to 25.5.

Based on these evaluations it is concluded that the final tuner is an improvement over the old tuner. The change from what was basically an general auto-tuner being applied in a Futhark-setting to a domain-specific tuner has provided improvements in both tuning speed and performance of the auto-tuned programs. While further improvements to the tuner are possible, it is felt that the tuner is currently fit to solve threshold-related auto-tuning tasks.

References

[1] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea, Finpar: A parallel financial benchmark, *ACM Trans. Archit. Code Optim.* **13**, 2 (2016), 18:1–18:27.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, Opentuner: An extensible framework for program autotuning, *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, *PACT '14*, ACM, New York, NY, USA (2014), 303–316.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, *IISWC '09*, IEEE Computer Society, Washington, DC, USA (2009), 44–54.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, A performance study of general-purpose applications on graphics processors using cuda, *J. Parallel Distrib. Comput.* **68**, 10 (2008), 1370–1380.

[5] M. Frigo and S. G. Johnson, The design and implementation of fftw3, *Proceedings of the IEEE* **93**, 2 (2005), 216–231.

[6] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI 2017*, ACM, New York, NY, USA (2017), 556–571.

[7] D. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, Chapter 3 - introduction to opencl, *Heterogeneous Computing with OpenCL 2.0*, Morgan Kaufmann, Boston (2015), 41 – 73.

[8] R. W. Larsen and T. Henriksen, Strategies for regular segmented reductions on gpu, *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, *FHPC 2017*, ACM, New York, NY, USA (2017), 42–52.

[9] C. Nugteren and V. Codreanu, Cltune: A generic auto-tuner for opencl kernels, *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip* (2015), 195–202.

[10] M. Tillmann, T. Karcher, C. Dachsbacher, and W. F. Tichy, *Application-independent Autotuning for GPUs*.

[11] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, A scalable auto-tuning framework for compiler optimization, *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, *IPDPS '09*, IEEE Computer Society, Washington, DC, USA (2009), 1–12.

[12] S. Tsutsui and P. Collet, Chapter 1 - why gpgpus for evolutionary computation?, *Massively Parallel Evolutionary Computation on GPGPUs*, Springer Publishing Company, Incorporated, Berlin (2013), 3 – 14.

[13] M. Vollmer, B. J. Svensson, E. Holk, and R. R. Newton, Meta-programming and auto-tuning in the search for high performance gpu code, *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, *FHPC 2015*, ACM, New York, NY, USA (2015), 1–11.

[14] R. C. Whaley and J. J. Dongarra, Automatically tuned linear algebra software, *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, *SC '98*, IEEE Computer Society, Washington, DC, USA (1998), 1–27.