UNIVERSITY OF COPENHAGEN

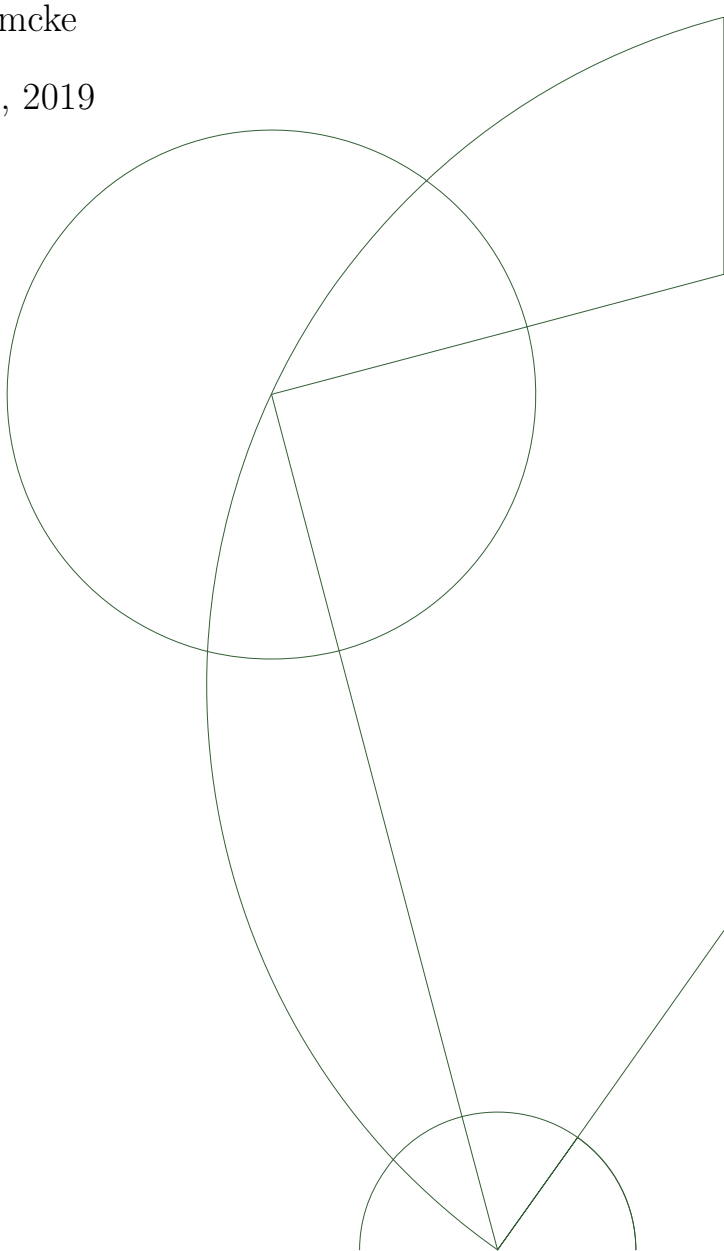
Master's Project:

Futhark Autotuners for Incremental Flattening

Svend Lund Breddam - QNB705

Supervisors:
Cosmin Eugen Oancea
Fabian Cristian Gieseke
Stefan Oehmcke

September 2, 2019



Abstract

This thesis details an automatic tuning strategy for threshold values in Futhark's incremental flattening compilation scheme. Incremental flattening is a compiler technique in which multiple semantically equivalent code versions representing differing code optimisation strategies are combined into one program. This allows individual input datasets to choose the optimisation strategy that best suits its characteristics given a particular hardware configuration. This flexible choice can be very useful in practice, but requires the tuning of a set of threshold values before the code is used in production.

The thesis at hand considers two cases of tuning strategies. The first one yields optimal threshold assignments with solid guarantees in a simplified setting. For the second one, four different optimisation strategies are employed and evaluated, including binary search, evolutionary strategies, active learning, and an experimental program instrumentation tuner. Each option provides its own benefits and problems, but they all show that tuning thresholds to improve the runtime performance is possible, with the best tuner yielding a 3.07x-speedup over the standard Futhark compilation into OpenCL code.

Keywords: Auto-parallelisation, hyperparameter tuning, functional GPU programming, code optimisation.

CONTENTS

CHAPTER 1

INTRODUCTION

General purpose computing on graphics processing units (GPGPU) is particularly interesting in the 21st century, as the growth of CPU single-core frequencies has stagnated, brought on by increasing thermal challenges of following Moore's law [1, 2, 3]. This, in addition to the growing interest in computing on Big Data, has allowed massively-parallel hardware such as GPUs to flourish [4]. They deliver a massive increase in core count with a reduced frequency, supported by a hardware architecture focused on executing a large number of smaller computations in parallel to produce results. On tasks that rely on a large amount of parallel computations, such as matrix matrix multiplication used in many computationally intensive algorithms, a GPU can massively outperform a CPU [5]. Due to the significant difference in hardware architecture, however, coding for GPUs has long been a very difficult prospect for programmers [6, 7, 1].

One modern approach for programming efficient GPU code is to rely on specialised programming languages and compilers [6, 8, 7], such as Futhark developed at the Department of Computer Science at the University of Copenhagen. Futhark is a statically typed, data-parallel, and purely functional array language with a heavily optimising ahead-of-time compiler for generating CUDA or OpenCL code [9, 1]. A core goal of Futhark is to combat the difficulty of coding on GPUs by hiding the challenges that comes with the GPU architecture behind its compiler. This in turn shifts the burden of knowing the intricacies of GPU development away from the programmer, and onto the compiler.

As part of ongoing research into programming languages, one technique for improving the generated code has been dubbed *incremental flattening* [10]. This compiler technique produces multiple semantically equivalent *code versions*, each corresponding to the application of different code-optimisation strategies driven by exploiting different amounts of application parallelism on hardware. Choices for code optimisation range from performing full flattening to register and block tiling [11, 12], but depending on the hardware and input dataset the improvement of each varies [10]. These *code versions* are then combined into one complete program by separating them with predicates, in the style of nested if-statements. These predicates compare the degree of parallelism that is mapped to the hardware in the current version to a *threshold*, allowing for the code version best suited for each dataset to be picked at runtime. Incremental flattening thus enables a flexible choice for each input dataset under the same compiled program.

The compiler code to generate these incremental flattening programs has been developed, whereas the subject of properly mapping dataset and hardware characteristics to code versions via the thresholds, is the main objective of this thesis. This will be achieved by *tuning* the thresholds guarding the code versions based on benchmarking information. The goal is to be able to perform this tuning automatically, such that given a target program and some supplied training datasets, the *tuner* will produce an assignment of threshold values, which yield optimal performance for each dataset on the given hardware, including unknown datasets.

This thesis will describe the process of solving this problem, starting with defining the incremental flattening setting. From there, multiple tuners using various optimisation strategies are implemented and evaluated on well known GPGPU benchmarks. This will be done incrementally, starting with the simplest case, and working up to a more difficult class of problems where data science techniques and fundamentals are necessary.

1.1 Introduction to Futhark and Flattening

While the goal of this thesis is to show that it is possible to map dataset characteristics to different code optimisations under specific hardware in general, the experimental work focuses on Futhark's incremental flattening compiler. The tag-line that the creators of Futhark most closely associate with their language is *"Why Futhark? Because it's nicer than writing CUDA or OpenCL by hand!"* [1].

As mentioned in the introduction, Futhark is a statically typed, data-parallel, and purely functional array language, with a heavily optimising ahead-of-time compiler. It is also strictly evaluated and supports in-place updates due to a uniqueness type-system ensuring race-free updates. In essence, Futhark follows most of the standards of a purely functional data-parallel language in the spirit of Guy Blelloch's NESL [13], with an emphasis on nested parallelism and parallel constructs.

While NESL was created before GPGPUs became mainstream, Futhark embraces it fully to make GPUs available to programmers. In order to grasp the motivation for making GPU programming easier, the following section will give a brief introduction to the GPU programming model. After that, we will showcase how Futhark code looks and how it is optimised, along with a proper explanation of how incremental flattening works in the context of Futhark.

1.1.1 GPU Programming

Two primary frameworks for GPGPU exists in OpenCL and CUDA. CUDA was first released in 2007, and is different from its OpenCL counterpart released in 2008 in that CUDA is vendor-specific to NVIDIA GPUs, which OpenCL is not. While they differ slightly in certain specifics, such as memory management [14], the two frameworks are extremely similar except for their nomenclature.

Both work on the same style of physical GPU layout, and have their programming models structured accordingly. Coding for GPUs is a matter of having the host (usually the CPU) offload certain computations onto a separate computing device (usually the GPU), and reading their results. In practice, this boils down to the following steps:

1. Memory is allocated on both host and device for input and output.
2. The input data for the computation to be offloaded is transferred from the host to the device.
3. The device performs the computation, under its massively parallel architecture.
4. The output from the device is transferred back to the host from the device.

For a programmer to implement this, he or she has to be conscious of the data transfers, and to both write orchestration code for the host, and the parallel code for the devices. Memory is slow in terms of performance, and for GPU programming this is especially important. In the small abstraction above, for example, transferring data between host and device is costly, and if the speedup is not greater than the overhead, then the computation should be kept on the host device.

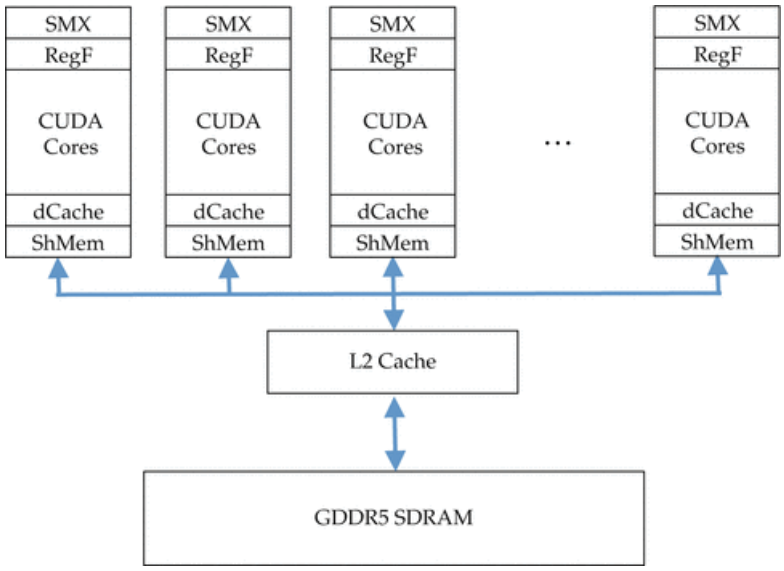


Figure 1.1: *NVIDIA GPU memory layout illustrated. In the figure, SMX stands for Streaming Multiprocessor. Source: Richard Vuduc [4]*

Figure 1.1 illustrates the typical memory layout of an NVIDIA GPU, but the structure is much the same for other vendors. The idea is that a GPU at the lowest level contains hundreds to thousands of functional units, called *Stream Cores* for AMD hardware and *CUDA Cores* for NVIDIA hardware. These processors are split into groups called *compute units*, which for AMD is called *SIMD engines* and *Streaming Multiprocessors* for NVIDIA. Comparing hardware between the two vendors can be problematic, but in essence they are structured in the same way [4, 14].

In terms of memory, there are three main levels in the hierarchy. Every compute unit has access to the same global memory, seen in Figure 1.1 as `GDDR5 SDRAM`, which is a common example of the type and protocol for data transfers used. Inside each compute unit, all functional units have access to the same shared local memory, and each functional unit also has private registers. Additionally, the GPU hardware contains caches between the different levels. Of these memory levels, host-memory, shared and global memory are left to the programmer, with the intermediate caches and registers being handled by the compilers [4].

Setting aside the memory aspects, the second difference comes in the programming model required to match the hardware architecture outlined above. The typical GPU architecture requires computations to be performed in a lockstep SIMD fashion, with SIMD meaning *single instruction multiple data* in Flynn's Taxonomy [15, 16]. SIMD execution refers to each of the functional units being given the same instructions to perform, but on separate data, with units executing completely in parallel.

As an example using OpenCL, each functional unit executes part of a *kernel* on its own data. These kernels are defined over the entire data to be processed, which is partitioned into separate *work group* grids, consisting of individual *work items*. Every functional unit executes one work item at a time from its work group concurrently alongside all other functional units inside that compute unit, meaning they can also be viewed as traditional multiprogramming threads. On simple 2D data such as a matrix, Figure 1.2 illustrates this, and it should be noted that communication between work items is only possible in their respective work groups. When a *kernel* is launched, all functional units across all workgroups are launched at the same time, executing the exact same function. Multiple work groups are thus run concurrently between the compute units available, meaning multiple levels of concurrency are used to orchestrate the compute power available in the GPU [14].

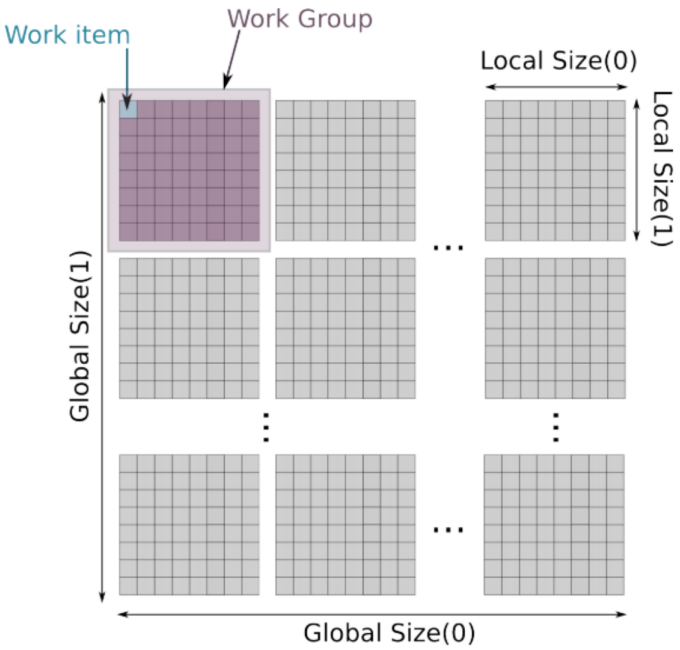


Figure 1.2: *Illustration of work items and work groups in an OpenCL kernel. Source: Tompson & Schlachter [14]*

The difficulty of programming for a GPU thus comes both from managing memory, and from working under a programming model very different from the traditional sequential one used by CPUs. This is particularly true since improperly managing memory transfers from global memory on GPUs is extremely costly, and mitigating this can require extensive rewrites of the code. Two such techniques, which can improve performance through proper memory management, are *tiling* and *coalesced accesses* to optimise locality of reference [12, 11].

Coalesced accesses of memory refer to the situation where functional units in a GPU access consecutive global memory locations in their SIMD load or store instructions [12]. The benefit of this is that the memory controller on the GPU can complete all 16 requests in a

single load transaction, which if not coalesced could result in up to 16 separate ones. This is essential to good GPU code, as memory becomes the limiting factor in many common use cases. Optimising spatial and temporal locality of reference using tiling to achieve coalesced accesses along with cutting down on accesses is one transformation worth understanding.

Tiling is a specific transformation in which memory accesses are reduced, by optimising for specific reuses of data patterns. This can be done both on block level with stripmining, and with registers by doing stripmining followed by loop unrolling and jamming [12, 11]. Loop-stripmining is a safe transformation in which a loop is turned into a perfect loop nest, one with a stride of T and one of stride 1. Stripmining is always safe, since it executes the same loop iterations in the same order as before. *Block Tiling* is a transformation where multiple loops are stripmined, and then having the loops of stride 1 interchanged inward to optimise locality, typically to achieve coalesced accesses. Another type of tiling named *Register Tiling* is achieved by first doing loop stripmining, followed by completely unrolling the inner loop. Unrolling is a transformation in which a loop is replaced by all the explicit iterations of the loop, cutting down on the control logic of the loop. While doing this unrolling, if any variable in the iterations of the original loop were invariant to that loop, their memory accesses will have been considerably reduced. Thus, register and block tiling are good optimisations to improve performance given recognisable data patterns, but which a novice GPU programmer might not know about, or find difficult to write.

While it is possible to get into the uses of GPGPU with the material available today for programmers, writing efficient code is still a major challenge. Additional challenges exist, such as taking into account the synchronisation of the many threads involved. All of these challenges underlines the necessity of automated tools for efficient GPU programming.

1.1.2 The Futhark Programming Language

As mentioned earlier, one of Futhark's design goals is to make GPU programming more available to the general programmer, by having efficient compiler techniques to achieve good code performance. In order to achieve this goal, the programming model adopted by Futhark is that of a purely functional data-parallel language. In order to illustrate this, consider the code example of matrix matrix multiplication shown in Listing 1.1.

```
1 let matmult [N][M][P] (x : [N][M] i32) (y : [M][P] i32) : [N][P] i32 =
2   map (\xr ->
3     map (\yc ->
4       reduce (+) 0 (map2 (*) xr yc)
5          ) (transpose y)
6       ) x
```

Listing 1.1: Matrix matrix multiplication in Futhark

The example starts with a type definition of the function, which takes as input two 32-bit integer arrays `x` and `y` of sizes $N \times M$ and $M \times P$ respectively, and outputs an $N \times P$ 32-bit integer array. The body of the function contains two outer maps, with an inner body containing a reduction of a pair-wise map. All of these operations are part of what Futhark calls *second order array combinators* (SOAC), which are operators on arrays that take as input a function to specify the operation of the SOAC.

The simplest example of this is `map`, which takes an array `x` and a function f, and applies f to every element of `x` independently. Likewise, `map2` is an extension of `map`, where

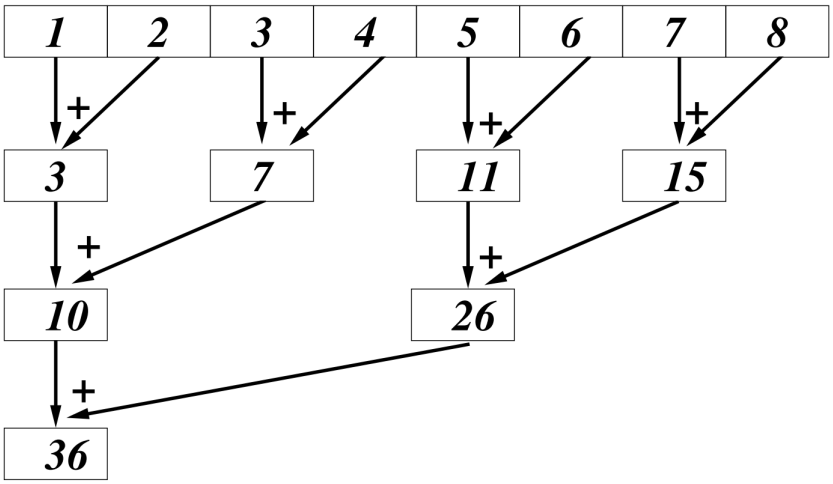


Figure 1.3: *Parallel execution of* `reduce` *to compute the sum on an array of eight integers. Source: Oancea [12]*

two arrays of the same sizes `x` and `y` are evaluated with the function f as a pair, yielding $f(x, y)$. Considering the layout of GPUs described in the previous section, a `map` is very intuitive to convert efficiently, as it can be seen as a single instruction applied to multiple data.

The other SOAC involved is `reduce`, which computes the aggregation of the input array according to the given function. The `reduce` function requires an array of type `[]t`, meaning an array whose elements are all of type `t`. It also requires an associative function f of type `t -> t -> t`, indicating a function taking two inputs both of type `t` to produce an output also of type `t`. Finally, `reduce` also requires the *neutral element* of the function f , which is the element n which satisfies the equation $f(n, x) = f(x, n) = x$. The evaluation of `reduce` combines all elements of the array using f, starting from the neutral element. The most basic example of `reduce` is to sum an array of integers, as it is shown in the code in Listing 1.2.

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

Listing 1.2: Computing the sum of an array in Futhark

The reason the function f has to be associative is to allow for efficient parallel execution. While `map` is intuitive to map to a work item in OpenCL, `reduce` requires coordination between work items in a work group to compute. Figure 1.3 shows a parallel execution of `reduce`, in which it is illustrated that $\log(n)$ parallel steps are executed in order to implement a reduction operation inside of a workgroup. While `map` is trivial to map to GPUs, `reduce` incurs a bit of overhead in comparison, as it is a more complex operation.

Returning to the code in Listing 1.1, the two outer maps extract the row of matrix `x` as `xr`, and the corresponding columns of `y` as `yc` by transposing `y` before the second map. The inner reduce then computes the sum over the array of pair-wise products of `xr` and `yc`. This version of matrix matrix multiplication is very easy to write with basic knowledge of programming in a functional language, and with the data-parallel nature is also easy for the compiler to convert into OpenCL or CUDA code.

The matrix matrix multiplication example was easy to write, and it has been explained

how the constructs involved translate to parallel GPU operations, but the goal of Futhark is to make this code efficient. Optimising the source code is done using a technique called *moderate flattening*, which serves as the base for *incremental flattening*.

Flattening Transformations

Flattening is a program transformation that eliminates nested parallel constructs, resulting in flat parallel data structure retaining all information about the nested structure. This transformation is important, since nested parallelism can not be directly mapped to the GPU architecture. Blelloch and Sabot [17] presented the transformation with a structure called a *field*, in which nested parallelism is expressed as a tuple of information describing the nest, and a flat array of values in said nest. The following example in Listing 1.3 exemplifies this notion in terms of nested irregular arrays, meaning arrays of arrays with differing row lengths.

```
1 [ [1,2], [3], [4,5,6] ] ==> ([2,1,3], [1,2,3,4,5,6])
```

Listing 1.3: Example of the flat parallel representation of an irregular array.

All the information about the original nested arrays is expressed in the new structure. The first part denotes the lengths of each *segment* in the original array, and the second part of the structure has the values of the original array presented as a flat array. Flattening extends to arbitrary depth, as storing multiple shape arrays corresponding to further nested data, while still keeping the data's values itself as a flat array.

In order to express programs on this changed data structure, the flattening transformation also applies to code itself. A set of rules for transforming nested parallel operators into flat parallel ones exist, which uses the above principles to expose more parallelism. One of the most intuitive of these rules is to transform nested maps, as seen in Listing 1.4 where `A` is composed as `(A_shp, A_val)` when flattened.

```
1 A = [[1,3], [2,4,6]]          ==>   A = ([2,3], [1,3,2,4,6])
2 map( \row -> map f row) A)    ==>   (A_shp, map f A_val)
```

Listing 1.4: Left hand side represents the normal nested-parallel code, applying some function f to every row of A. The right hand side represents the same code, but using the flattening transformation.

How does a transformation like the one in Listing 1.4 help improve parallelism in arbitrary code? The transformed code does the same amount of work, using one single map on the flattened data rather than one map per row in `A`. We say that the flattening has *uncovered* additional opportunities for parallel execution, in that this transformed code is more easily mapped to the GPU.

A second example of a flattening rule is that of a `scan` inside of a map. A `scan` is a parallel operation to compute the prefix sum of an array, though being a SOAC it takes as input a function to compute said prefix sum with. Whereas a nested map was simple, a scan on the flattened value array requires a notion of when segments start and end. For this, the notion of a flag array is introduced, which is an array of the same length as the flat value array, with boolean values indicating the start of segments. With this flag array, a nested scan can be converted as shown in Listing 1.5

```
A = [[1,3], [2,4,6]]           ==>   A = ([2,3], [1,3,2,4,6])
map( \row -> scan f e row) A) ==>   (A_shp, sgmScan f e A_flg A_val)
```

Listing 1.5: Left hand side represents the normal nested-parallel code, scanning every row of `A` using f and it's neutral element e. The right hand side represents the same code, but using the flattening transformation.

Computing the flag array is also possible to do in parallel as a combination of other parallel operators, and allows for segmented operations such as that for scan. Segmented operations are operations which uses the flag array to do work on segments in parallel, such as computing the prefix sum across all rows in the same execution, rather than having the scan mapped across multiple kernels.

The flattening transformations thus allow for multiple compositions of parallel operators to be modified to work on flat data instead. Flat data is very useful in terms of parallelism, as it allows better utilisation of the hardware in a given kernel. Futhark incorporates these ideas into it's code generation through a technique named *moderate flattening*.

Moderate Flattening

Blelloch showed in 1995 with the development of NESL that flattening transformations could be built into a programming language, to allow for efficient data-parallel computations to be expressed [13]. This style of languages became known as data-parallel languages, which roughly follows the same programming model as NESL. Since this development, the idea of flattening data to fit onto massively parallel hardware, such as vector machines or GPUs, has been adapted into Futhark with moderate flattening.

The main drawback of doing full flattening, in which all nested parallelism is flattened completely, is that GPUs has a capacity for how much parallelism can efficiently be utilised. While full flattening on a theoretical machine has nice properties, it does not translate well to the real world where communication and memory costs are central to good GPU code performance. Using full flattening on GPUs is complicated further by it not being able to exploit locality of reference, which can considerably improve performance in many programs.

The question then becomes how to partially flatten a program to a point in which it completely saturates the GPU hardware available, but stops short of oversaturating it. In Futhark, this is implemented with *moderate flattening*. Moderate flattening works by matching compositions of parallel constructs against a number of flattening rules in order to convert nested parallel operations into sequences of parallel operations [9].

This algorithm results in a modified program, which has been restructured to allow for better utilisation of the parallel operations, since nested parallelism can not be directly mapped to the GPU architecture. More specifically, it reorganises the imperfectly nested parallelism into perfect SOAC nests, in which the outer levels correspond to map operators, while the innermost one is some arbitrary SOAC or scalar code. This reorganisation allows easier mapping to GPU architectures, as the map operator is easily representable on the GPU. This is done primarily using loop interchange and map distribution, while also trying to avoid more computationally expensive operations such as optimising parallelism inside if-statements, which would require filter operations.

As an example, consider the program distribution example in Figure 1.4. The code in 1.4

(a) contains multiple instances of nested parallelism, both inside maps and inside sequential loops. After distribution the code in 1.4 (b) only ever has maps on the outer level of nested parallelism, resulting in multiple examples of perfect nests capable of being mapped to GPUs.

```
let (asss, bss) =
map
  (λps: ([m][m]int,[m]int) →
   let ass =
   map (λp: [m]int →
      let cs =
        scan (+) 0 (iota p)
      let r = reduce (+) 0 cs
      let as = map (+r) ps
      in as)
     ps
   let bs =
   loop (ws=ps) for i < n do
      let ws' =
      map (λas w: int →
         let d = reduce (+) 0 as
         let e = d + w
         let w' = 2 * e
          in w')
        ass ws
      in ws'
   in (ass, bs))
 pss -- pss : [m][m]int
```

(a) Program before distribution.

```
let rss =
map (λps: [m]int →
   map (λp: int →
     let cs = scan (+) 0 (iota p)
     let r = reduce (+) 0 cs in r)
   ps) pss
let asss =
map (λps rs: [m]int →
     map (λint (r) →
       map (+r) ps)
      fs) pss rss
let bss =
loop (wss=pss) for i < n do
   let dss =
    map(λass: [m]int →
       map(λas: int →
          reduce (+) 0 as
      , ass), asss) in
    map (λws, ds: [m]int →
       map (λw d: int →
          let e = d + w in
          let w' = 2 * e in w')
       ws ds) wss dss
```

(b) Program after distribution.

Figure 1.4: *Example of the ideas behind the flattening engine in Futhark. The code is not quite Futhark code, but the parallel constructs and their distribution is the same.*

This reorganisation into perfect SOAC nests is done using a set of flattening rules conceptually reminiscent of Blelloch's work, with the major distinction being the goal of exploiting top-level parallelism rather than fully flattening from the bottom up. From said perfect nests, GPU code kernels can be extracted and form an optimised program.

While moderate flattening does improve the degree of statically exploitable parallelism, it still has one shortcoming. It only produces one optimised program for all its input datasets and hardware, depending on a chosen heuristic [9].

Having a single program designed for multiple input datasets is an example of *one-size-fits-all* optimisations, in which one choice of optimisation is chosen for all inputs. Most datasets vary in the amount of parallelism they offer however, meaning the degree of parallelism utilised by the hardware for any given optimisation varies greatly between datasets [10].

Consider the matrix matrix multiplication code from Listing 1.1 discussed earlier as the function applied to some input dataset. For a large input dataset, it is possible that only exploiting the outermost level of parallelism best saturates the hardware, while for a small dataset it might be possible to saturate the program using full flattening. If one were to only choose one of those two strategies, then the other dataset will suffer. Since input is by definition known only at runtime, this choice can not be accomplished by the static heuris-

tic. Inaccuracies from these heuristics can lead to severe underutilisation of the hardware available, again hurting performance [9, 10].

A modification of moderate flattening named incremental flattening aims to solve the above problem, by adding additional flattening rules and modifying some of the existing ones from moderate flattening. The general idea being that instead of making a single choice of heuristic, incrementally flatten parts of the code to produce multiple options for optimisation strategies, and choose from them at runtime.

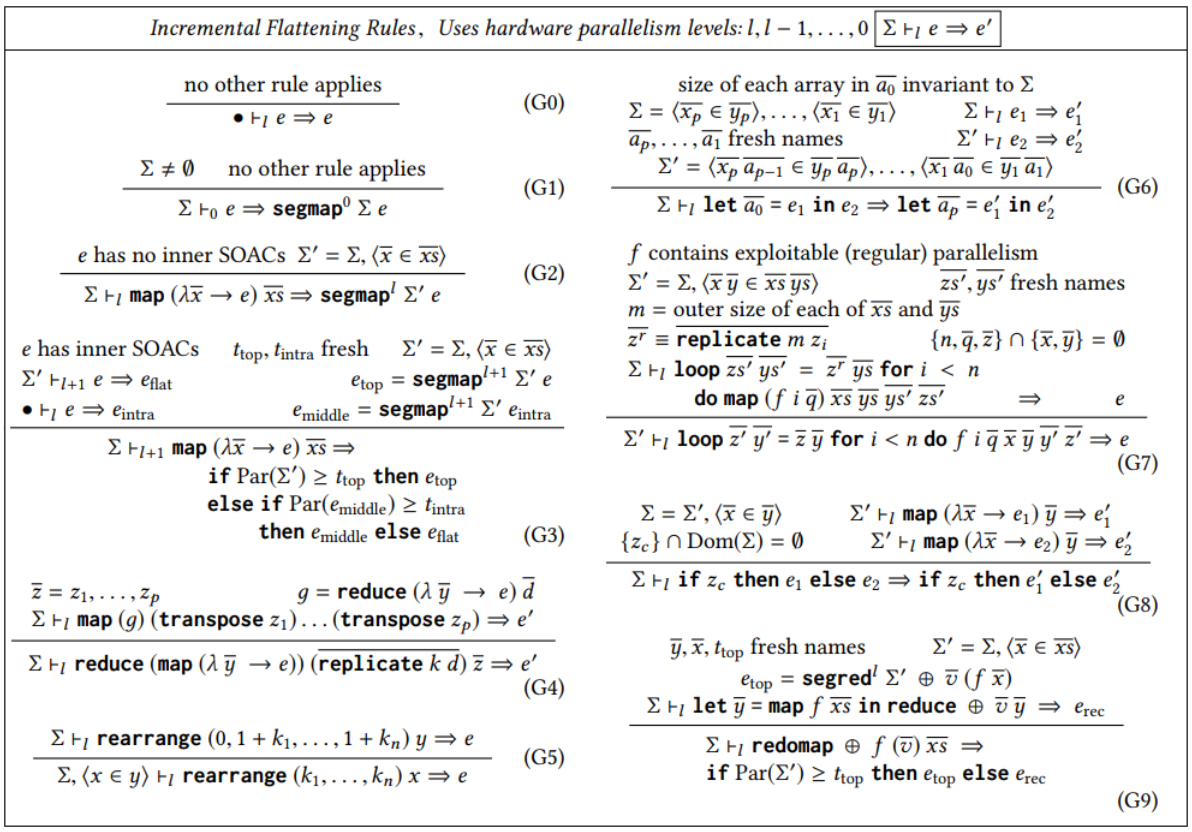


Figure 1.5: *Inference rules for the incremental flattening implementation in Futhark, as described by Henriksen, Thorøe, Elsman & Oancea [10]*

1.1.3 Incremental Flattening

The full inference rules for incremental flattening seen in Figure 1.5 aim to guide the existing moderate flattening algorithm to generate multiple code versions, each corresponding to different levels of flattening. Incremental flattening is a recursive flattening algorithm, with the first three rules (G0 - G2) representing the base cases, and the remaining rules constituting part of the original moderate flattening rules for how to perform flattening.

Rule G3 is the primary driver behind the multi code versioning. For each map in the target program that contains nested recurrences, G3 produces three code versions separated by predicates. e_{top} parallelises the map along with the earlier nested parallelism up till this point, and sequentialises the body. e_{middle} parallelises the map and recursively also the body, to utilise the inner parallelism as well using a different hardware level. In this explanation, hardware levels refer to the structure of the GPU, with optimisations happening at grid level (level 1) or work group level (level 0). The last version, e_{flat} attempts to continue flattening

on the same hardware level.

The remaining rules, G4 - G9, represent the different rewrites used during moderate flattening that are kept for incremental flattening. These specify the prerequisite patterns in the code with which specific rules apply to reorganise the code. One such example is rule G4 taken directly from moderate flattening. It represents map-reduce interchange, in which a reduce nested in a map in a specific way can be rewritten more efficiently. Taken together, all the rules from Figure 1.5 describe the code version generation.

Applying incremental flattening results in a combined program where each of the generated code versions is guarded by a predicate also generated in Rule G3. This predicate compares a free variable *threshold* against an expression for the maximum degree of parallelism available to be utilised in that version, seen in Figure 1.5 as "$Par(x)$". These predicates allow the combined program to choose different levels of optimisations dynamically, effectively bridging the gap between one-size-fits-all optimisations and different hardware and dataset characteristics.

Incremental Flattening Example

Returning to the matrix matrix multiplication example shown in Listing 1.1, when compiled using incremental flattening, five code versions will be produced, each utilising a different amount of GPU parallelism. First, the outer map is encountered which contains nested parallelism inside, being both the inner map and the reduce. This generates three code versions, two of which will reach the base case and finish. The third one will try to continue flattening further inwards, which encounters the inner map with a nested reduce. This one also produces three code versions, yielding a total of five. These five code versions are separated by predicates, which is shown alongside the matrix matrix code in Figure 1.6.

In order of when they are considered in the combined program, each version launches a different number of threads to handle different workloads. Version one (V1) will execute the body of the map sequentially, meaning every GPU work item will work on a single row in the array, launching N work items. Version two (V2) will try to partially parallelise the body, assigning one workgroup per row, launching $N \times B$ work items where B is the workgroup size.

Version three (V3) parallelises the second map as well as the first, but keeps the dot product sequentialised, launching $N \times P$ work items, each computing the dot product for one result element in the output. Version four (V4) again attempts to partially parallelise the dot product, assigning one work group per element, resulting in a total of $N \times P \times B$ work items launched. Finally, version five has fully flattened the code, in which $N \times P \times M$ work items are launched, each computing one multiplication in the dot products.

These five code versions are then separated by predicates as in Figure 1.6, which will compare the degree of parallelism against the free variable thresholds. In addition to these predicates, additional hardware-constraining predicates are also introduced in order for specific code versions to run. One such example is e_{middle} that tries to assign one workgroup, which will only run if this assignment is possible with the given hardware constraints.

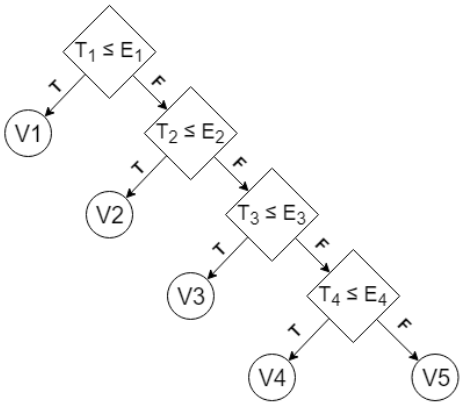
After deciding on these code versions, the incremental flattening compiler performs additional simplification, kernel extraction and access pattern optimisations. This includes code to achieve coalesced accesses, or to use tiling to improve locality, as mentioned earlier. It

(a) Code for matrix matrix multiplication in Futhark

```
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
```

```
if (Threshold 1 <= E1_par(Input)):
    Execute code version V1
else if (Threshold 2 <= E2_par(Input)):
    Execute code version V2
else if (Threshold 3 <= E3_par(Input)):
    Execute code version V3
else if (Threshold 4 <= E4_par(Input)):
    Execute code version V4
else:
    Execute code version V5
```

(b) Predicates for incremental flattening of (a).



(c) Tree structure of predicates in (b).

Figure 1.6: *(a) Futhark code example of matrix-matrix multiplication from Listing 1.1. (b) Pseudo code for combined predicate guards. (c) Tree structure for the predicate guards. All five code versions are semantically equivalent.*

also looks for other patterns which leads to speedups, such as the `redomap` construct used in many benchmarks [18].

Incremental Flattening's Tree Structures

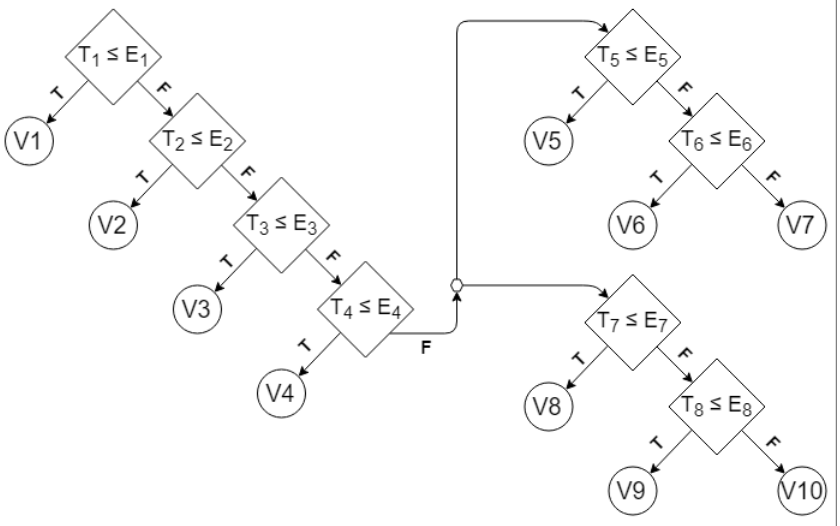
Through incremental flattening, the code versions can be chosen at runtime based on the predicates using threshold values. For the tuning processes later, this combined program can be represented using a tree structure, since the predicates are ordered such that there are inter-version dependencies. For this tree, nodes represent a predicate guard of the form $Threshold \leq E_{par}(Input)$, where $E_{par}(Input)$ is an expression of the maximum parallelism used by an optimisation on the given input dataset for this code version, and leaf nodes represent a single chosen code-version. Due to the structure generated by Rule G3 of incremental flattening, most of these predicates are nested, and thus have dependencies between them. In the tree, this is represented by every node being dependent on the evaluation of all previous nodes. Figure 1.6 (c) shows the earlier matrix-matrix multiplication's five code versions decided by the nested predicates.

This predicate structure is complicated by two specific situations. First, it is possible for multiple nodes to be dependent on the same parent node, resulting in a non-binary tree structure. In the program, this represents optimisations of the same depth in nested parallelism. An example is the code from the LocVolCalib benchmark, which in simplified form can be seen in Figure 1.7 along with a tree structure. It has a map containing two nested map each calling the "TriDag" function with two different inputs. Since these are on the same *depth* of the nested parallelism, their optimisations rely on earlier choices for the outer map, but both can be optimised in different ways due to their $E_{par}(Input)$ differing.

(a) LocVolCalib simplified Futhark code

```
map2 (\ xss0 yss0 ->
  loop xss , yss = xss0 , yss0
    for t < numT do
      let xss ' = map tridag xss
      let yss ' = map tridag yss
      in ( xss ' , yss ')
  ) xsss0 ysss0

-- ( xsss0 : [ numS ][ numX ][ numY ] f32 )
-- ( ysss0 : [ numS ][ numY ][ numX ] f32 )
```



(c) LocVolCalib tree structure

Figure 1.7: *(a) Simplified Futhark code of the structure in the LocVolCalib benchmark. Source: PPoPP19 [10]. (b) Resulting incremental flattening tree structure, following the legend present in Figure 1.8*

The second more challenging situation comes into play when $E_{par}(Input)$ has input which changes size multiple times during execution. This happens when executing nested parallelism code inside a sequential loop, and the size of the input array of the nest changes sizes with each iteration of the loop. This is a difficult situation, since only one threshold value is now compared against multiple different $E_{par}(Input)$, as opposed to having only one comparison against it made. This does not influence the tree itself, but proves a challenge on figuring out a proper value for the thresholds.

As an illustration of when this case happens, Listing 1.6 contains the code of one instance of the problem. The code in Listing 1.6 performs matrix matrix multiplications of two matrices, but in each iteration of the loop the shapes are changed by $log(n)$ according to the loop. Having this specific pattern results in code versions having multiple values of $E_{par}(Input)$ to choose from. Since this code essentially performs matrix matrix multiplication inside of a sequential loop, the tree structure is identical to that of the earlier matrix matrix multiplication code seen in Figure 1.6 (c), though with all four thresholds affected by the sequential loop problem. While it is not solely the presence of a sequential loop that creates this class of problem, in the rest of this thesis this difficult situation is dubbed that of the *loop based trees*.

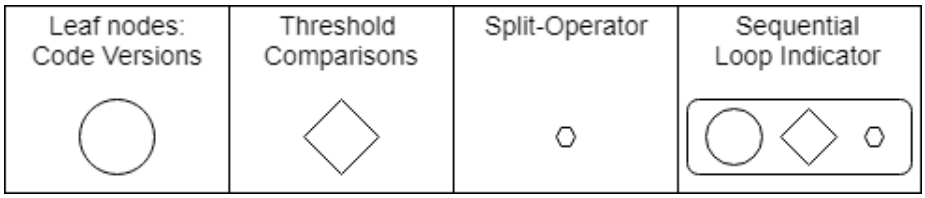


Figure 1.8: *Legend for future branching trees.*

```
let main [n] (A : [n][n]f32) (B : [n][n]f32) =
  let ks  = map (\i -> 1<<i) (iota (1i32 + (lg n)))
  let M   = []
  let (res, _) =
    loop (M,A) for k in ks do -- k = 1, 2, 4, 8, 16, 32, 64, 128
      let A' = unflatten (n/k) (n*k) <| flatten A -- size of A' is n/k_
          {i-1} x n/k_{i-1}
      let B' = unflatten (n*k) (n/k) <| flatten B
      let M  = matmul A' B' -- n/k x n/k
      let A''= map2(\Arow Mrow ->
                    map(\j ->
                        if j < n/k
                        then Arow[j] + unsafe Mrow[j]
                        else Arow[j]
                    ) (iota (n*k))
                ) A' M
      in  (M, A'')
    in  res
```

Listing 1.6: Futhark code of a sequential loop with variant sizes

This tree structure will prove useful for any non-black box tuning technique, either to implement caching of results, or to otherwise navigate control flow of the combined program. In the future, tree-graphs will use the legend in Figure 1.8 to incorporate the above complications in their representations. The split-operator hexagon denotes the first situation, where a sequence of logical OpenCL kernels happen at the same depth of nested parallelism. Lastly, the box indicates parts of the tree executed in a sequential loop, where the degree of parallelism in the input is influenced by the loop, resulting in a threshold having multiple values of $E_{par}(Input)$ compared against it.

CHAPTER 2

FUTHARK AUTOTUNER IMPLEMENTATIONS

Futhark's Incremental Flattening produces a program with optimisations exploiting varying degrees of available hardware parallelism. The goal of the autotuner is to choose threshold values, such that each training dataset executes the code version best suited for it [10]. This has to be achieved without prior knowledge of which datasets prefer which code version, and without knowledge of future datasets being introduced. From these training datasets, the thresholds chosen should generalise such as to accurately cluster future unknown datasets to good code versions.

In terms of an optimisation problem, the objective function to be minimised reflects the runtime of each dataset and the inputs are the tuple of chosen threshold values $(T_1, T_2, \cdots, T_N)$ along with the target program. Additionally, while solving this problem, an efficient solution minimises the number of benchmarks to run, since that is the primary way of getting information about the thresholds' optimal values, but takes the most time of the tuning process.

All the tuners' primary source of information is Futhark's native benchmarking tool, which makes it possible to run a program with all its given datasets under a supplied threshold configuration. The information gained per run is each dataset's runtimes, and the values of $E_{par}(Input)$ that thresholds were compared against for that dataset [1]. In particular this last part is useful, as it contains a lot of information about how the threshold values will alter the control flow from one code version to another, following the program's tree structure.

As an example, if a threshold is set to 400 then all $E_{par}(Input) < 400$ will choose the same version, meaning any two values in that range are essentially identical in terms of performance for that dataset. Similarly any two values chosen greater than 400 would both choose the other available code version in that comparison. Any autotuner implementation will have to exploit this to efficiently find the optimal thresholds without repeating experiments that yield no new information.

The following sections will cover concrete implementation strategies for solving the tuning problem. Section 2.1 will cover the simplest case of tuning, whereas Section 2.2 will introduce the more difficult situation, in which advanced optimisation techniques have to be applied. Those more advanced solutions are presented in Chapter 3.

In terms of the tree structure from earlier, the simple case covers all trees except those

containing sequential loops in which the degree of parallelism inside of the loop varies within the loop. These are considered "*simple*" since it will be shown that a recursive algorithm can efficiently solve the problem and provide decent guarantees that do not hold for the difficult case.

2.1 Base Futhark Autotuner

For the simple case, programs will never contain a sequential loop, and only ever have one value of $E_{par}(Input)$ per threshold for each dataset. This is a major simplification in terms of how well we can produce good values of thresholds. Having this simplification results in a lot of certainty about the control flow of the program produced by Incremental Flattening when examining the predicate guards. In terms of the tree structures, the first two trees in Figure 2.1 (a) and (b) would fall under this simplification, while the last one in (c) will be dealt with later in Section 2.2 and Chapter 3.

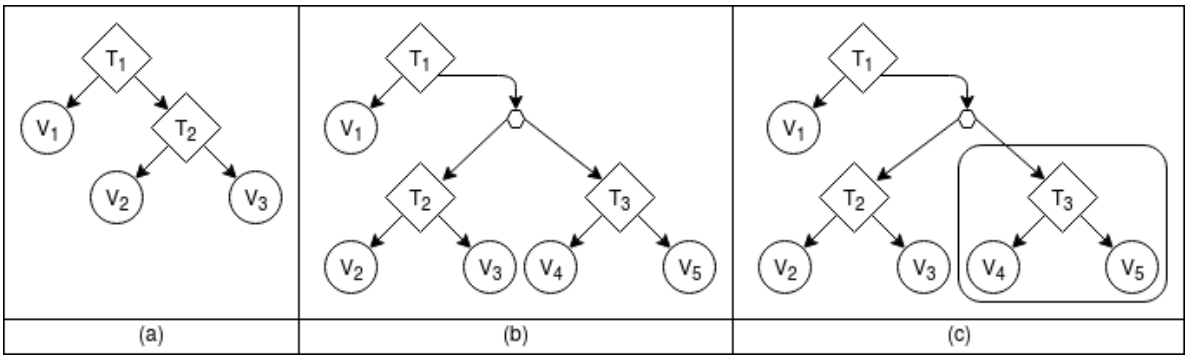


Figure 2.1: *Three types of trees: (a) has no complications involved. (b) has two branches of optimisations at the same depth. (c) has parts executed in a sequential loop.*

We start by defining the useful concept of a *threshold range*. As mentioned earlier, for threshold comparisons against a single value of $E_{par}(Input)$, any threshold choice below that value will yield the same code version due to the predicate's form of $Threshold \leq E_{par}(Input)$.

To return to the earlier example with $E_{par}(Input) = 400$, if a dataset were to prefer the predicate evaluating to true, then the range of threshold values that would satisfy that constraint would be $(1, \cdots, 400)$, and if it preferred the predicate being false then it would be $(401, \cdots, \infty)$. This range represents all the *optimal* values for a single threshold under a single dataset, depending on which version was preferred. The goal of finding the best threshold values can now be reformulated as finding the threshold ranges satisfying all datasets.

The simple algorithm for finding these ranges is shown as pseudocode in Algorithms 1 and 2 with most of the work done in the latter. The former simply calls `OptimiseSimpleTree` from Algorithm 2 on every tree in the program, tuning each tree independently.

Algorithm 2 works by looping over all thresholds in the given tree $\mathcal{T}$ in depth first traversal. For each threshold T, every dataset has it's $E_{par}(Input)$ for that threshold comparison extracted in the function `GetValueOfE`, and using that information two benchmarks are run in `BenchmarkChoices`. The `BenchmarkChoices` function runs a benchmark with T set to 1,

and one set to ∞, resulting in both options in the threshold comparison node being tested for this dataset. The function then returns the threshold range for T with which the dataset would pick their preferred code version.

Once these threshold ranges are computed for every dataset, they are all intersected. If this intersection is not the empty set, that means a range satisfying all datasets is found, and the largest value in that range is chosen as the final optimal threshold. When the depth-first traversal has finished, the tree has been entirely tuned, and Algorithm 1 can tune the next tree in the program.

Algorithm 1 OptimiseProgram

Input: Program P with trees $\mathcal{T}^1, \cdots, \mathcal{T}^K$, and Datasets $D_1, \cdots D_M$.
Output: Optimal threshold assignments $(T_1^i)^*, \cdots (T_{N_i}^i)^*$ for each tree $\mathcal{T}^i$.

Trees = { }

for all trees $\mathcal{T}^i$ **do**
    Trees[i] = `OptimiseSimpleTree`$(\mathcal{T}^i, (D_1, \cdots, D_M))$
end for

Return: Trees

Algorithm 2 OptimiseSimpleTree

Input: One tree $\mathcal{T}$ with threshold names $T_1, \cdots, T_N$, and Datasets $D_1, \cdots D_M$.
Output: Threshold assignments $T_1^*, \cdots, T_N^*$ for the N thresholds in $\mathcal{T}$

opt_thresholds = { }

for all thresholds T_i in `DFS`$(\mathcal{T})$ **do**
    ranges = [ ]

    for all datasets D_j in $D_1, \cdots, D_M$ **do**
        $E =$ `GetValueOfE`(D_j, T_i)
        $[T_{lower},\ T_{upper}] =$ `BenchmarkChoices`$(\mathcal{T},\ T_i,\ E,\ D_j)$
        ranges.`append`($[T_{lower},\ T_{upper}]$)
    end for

    $[(T_i^*)_{lower},\ (T_i^*)_{upper}] =$`intersect`(ranges)
    opt_thresholds[i] = $(T_i^*)_{upper}$
end for

Return: opt_thresholds

Pseudocode Discussion

The pseudocode represents a depth-first recursive algorithm, where in each iteration each node's two choices are already "optimal", since the algorithm starts tuning from the deepest most level of the tree. If that comparison node can be made optimal, then the level above can also choose its optimal choice, and so on recursively. This pseudocode produces

a threshold configuration in which every threshold is set to the maximal empirical degree of parallelism, with which every training dataset chooses its optimal version. This works under the assumption that $E_{par}(Input)$ accurately depicts the degree of parallelism achieved by each code version, and that the code versions actually can be mapped to each dataset under specific hardware.

To show how the pseudocode intersections would work, consider the deepest comparison node in the Futhark Matrix Matrix multiplication code whose tree was shown in Figure 1.6. That node chooses between two code versions decided by a single threshold T_4. With an example dataset A, the predicate deciding between the versions was found to be $T_4 \leq 400$, and A preferred the version where the predicate evaluated to False. We now know any value of T_4 in the range $[401, \cdots, \infty]$ would result in A choosing the correct version. If another dataset B for the same node then had the predicate $T_4 \leq 800$, and we learned that B's optimal code version was the one achieved by the predicate evaluating to True, then we know any value of T_4 in the range $[1, \cdots, 800]$ would work for B. With this information it is possible to intersect those two ranges, such that both datasets A and B choose correctly, which in this case would result in the range $[401, \cdots, 800]$. This would be the optimal result, as A's predicate would evaluate to False, and B's predicate would evaluate to True, under the same threshold.

The example makes it clear that the final ranges will produce an optimal threshold configuration for all training datasets. If the intersection of all the separate threshold ranges is not the empty set, that means that new range has to also be a subset of all the individual ranges, and thus optimal for all datasets.

As for an actual implementation of the pseudo-code, it has to be considered that the optimal intersected range can be empty. This can happen either because E^j doesn't accurately reflect the degree of parallelism, or because variances in benchmarking calls could incorrectly identify a near-optimal code version as optimal. To break this stalemate, one approach is to leverage the information that has previously been recorded while constructing the ranges.

The intersection algorithm can be reworded as finding the the minimum T_{upper} value, and likewise finding the maximum for the "lower" values, as in Equation 2.1 where N is the number of datasets.

$$T^* = [T^*_{lower}, T^*_{upper}] = [\texttt{max}(T^1_{lower}, \cdots, T^N_{lower}),\ \texttt{min}(T^1_{upper}, \cdots, T^N_{upper})] \tag{2.1}$$

This intersection is *empty* if $T^*_{lower} > T^*_{upper}$, but if it is empty we still know the values of the lower and upper T^*. The heuristic we chose to break these stalemates uses these two values to retry the benchmarks with the threshold set to either of those two values merged values, and picking the best one. This works particularly well when the majority of datasets prefer one of the two, and only a few datasets caused the intersection to not be empty. It does have the downside of favouring long-running datasets if the function for choosing between the two is accumulative runtime.

Accumulative runtime is a decent measure of overall performance, with the one flaw that it favours longer running datasets. Problems arise with this if all datasets, regardless of size, are of equal importance. In such instances, it can be remedied by implementing a weighted sum. One way of implementing this is to record a baseline performance of all datasets, and then have the weighted sum reflect the percentile improvement. However, this potentially favours shorter running datasets, in which noise can have a large percentage influence.

As for some other smaller details of the algorithm, they require explanation as well. Algorithm 1 loops over multiple trees $\mathcal{T}$ in the program, which refers to a program having multiple separate instances of nested parallelism in its code. Looping over them sequentially is done in order to keep all thresholds not relevant to a specific tree $\mathcal{T}$) fixed, in order to isolate one threshold's impact at a time and thus reduce overall tuning time considerably.

Secondly for the inner workings of the function `BenchmarkChoices` in Algorithm 2, it is assumed that all thresholds above T_i in the tree $\mathcal{T}$ are set to ∞, and all thresholds below T_i set to their optimal value found earlier. This ensures that only the current threshold T_i influences the result, and that it chooses between two potentially optimal choices. Having all thresholds be false to start is in line with base incremental flattening without tuning, where every threshold is set to 2^{15} [10].

Finally, when fixing a single threshold value from the final range in Lines 12-13 in Algorithm 2, it is chosen as the maximum allowed value, $(T_i^*)_{upper}$. It is possible to return the computed ranges to choose from, but to actually run the program with the thresholds a singular value has to be chosen, and T^j_{upper} reflects the maximal empirical degree of parallelism the code version in that comparison can effectively use. Choosing the final value like this also helps generalise better to unknown datasets.

Because the T^j_{upper} was computed from experiences, given a future dataset with an $E_{par}(Input)$ greater than T^j_{upper}, we know that we earlier had a training dataset whose $E_{par}(Input)$ was also greater than T^j_{upper} and preferred the predicate evaluate to False. Due to this, the unknown dataset should ideally follow the same pattern, and thus choose the correct version as in our earlier experience.

Incremental Flattening's shortcoming

With the pseudocode described and discussed, there is one relevant shortcoming of Incremental Flattening regarding empty intersections. Below in Figure 2.2 is pseudo code with two sample datasets, which under a specific optimal optimisation strategy will never be solved by Algorithm 1 for both datasets. The code has a map over some inner parallelism, with the map working on a variable of size N and the inner parallelism working on a size of M.

```
1  map (\x ->
2          inner_parallelism(M)
3      ) (iota N)
4
5  -- Datasets
6  -- D1 : (N, M) = (100K, 20K)
7  -- D2 : (N, M) = (200K, 1024)
```

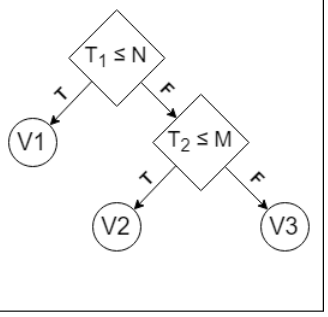


Figure 2.2: *Pseudocode example of an unsolvable case, with it's simple tree structure.*

From the definition of Incremental Flattening the program compiles into three code versions, one for sequentialising the inner parallelism, one for mapping the inner parallelism at workgroup level, and a fully flattened version. With the code above, it is impossible to choose

thresholds to implement the following strategy of optimal code versions:

If the size of the inner parallelism fits the max workgroup size, then always execute the work group version, otherwise execute the outer map in parallel and sequentialise the rest.

If we produce two datasets with varying N and M values as listed above, and assume $E_{par}(Input)$ is simply the values N and M for the two comparisons, we will have a conflict with the strategy. The crux of the problem is that the choice for best version in the strategy is decided by M which is compared after N due to rule G3's ordering of comparisons from Figure 1.5. First the comparison to N is made, which will decide whether to sequentialise the inner parallelism, without having yet compared M to know if we in fact have sequentialised it or not.

Fixing this is not in the scope of this project, and would require some revisions of the Incremental Flattening algorithm itself, with either a rework such that the first comparison also takes M into account, or to switch the two code versions priority in Rule G3 which produces the ordering. What this means for this project is that tie-breaking will be a possibility in practice, in that some programs might not have an optimal assignment.

Guarantees

Due to the simplification of having a certainty about the control flow in the combined incremental flattening program, certain guarantees can be shown. In particular the following guarantee is essential:

"If an assignment of thresholds exists which satisfies all datasets, Algorithm 2 will produce said optimal assignment using one benchmark run per code version in the program."

As it states, it only holds in the case where there is in fact a solution for all datasets, which as just covered is not entirely guaranteed. In those cases where no such solution exists, it does require only two additional benchmarks to be run in order to find a near-optimal assignment per conflict, using the heuristic described earlier. In order to achieve the guarantee above in practice, caching has to be employed to save the results of previous nodes.

Consider a tree with two comparison nodes separating three code versions, like the one in Figure 2.2. To solve the bottom most comparison node, the two benchmarks run in `BenchmarkChoices` makes it possible to compute the resulting runtimes of every possible threshold assignment, for every single datasets. This comes again from the certainty of control flow, and will be referenced as *complete information*.

Returning to the idea about caching, consider Algorithm 2 when it tries to solve some comparison node which isn't the bottom most node. First, it benchmarks the new unseen code version, and then it benchmarks the false predicate choice. Using the reasoning above, the false predicate choice's runtime can be computed without running any additional benchmarks, as 2 already has complete information about the nodes below. With this, the number of benchmarks run is reduced from two per comparison node down to one per code version, which gives us the guarantee.

Both the *complete information* aspect and the argument for it's output being optimal comes

from the threshold range concept. Attaining *complete information* in the simple setting is not difficult, but as we will soon move on to loop based trees the property is much more difficult to attain.

To wrap up Algorithm 1 for the simple case, if all the threshold ranges are easily intersectable, this tuning strategy will find the correct result in X attempts, where X is the number of code versions in the incremental flattening program. If this is not the case, or the program contains a sequential loop with varying parallelism inside as described in the Futhark introduction section, then additional work has to be done.

While Algorithm 1 is unable to solve the more difficult loop based trees, it will still serve as a base for algorithms solving those trees, since not all nodes in the difficult trees are difficult, as for example the third tree shown in Figure 2.1 (c). The simple case algorithm shows that elegant solutions to the tuning problem can be found in the case of Futhark, and the following sections aim to extend this to the last type of trees.

2.2 Loop Based Trees

As opposed to the simple case, having multiple values of $E_{par}(Input)$ per kernel and dataset pair necessitates a different approach to the tuning problem. Previously it was possible to gain all the necessary information in a few benchmarks, while having arguments about the correctness of each threshold assignment. Under the loop-based trees, the same code version is run multiple times on different inputs, with different parallel sizes, but still decided by a single threshold value. If this piece of code is run once pr. iteration of the sequential loop, the number of $E_{par}(Input)$ values equals the number of iterations of the loop, which varies greatly from program to program. The role of the threshold thus shifts in comparison to the simple case, in which a singular code version had to be chosen as optimal based on the threshold. Its new role is to distinguish between two code versions, and to choose the best in each iteration of the sequential loop, all with a singular threshold value. The main difficulty of this case comes from not having the runtimes pr. sequential loop iteration, but having the runtime as the summation of all iterations.

One way to look at this more difficult problem is as an optimisation problem, in which we aim to find the minimum value of the function $f(\bar{T})$ where f is the accumulative runtimes of datasets in a benchmark run using $\bar{T}$, and $\bar{T}$ is the threshold configuration supplied. From this point of view, a varied range of options become available for tackling the problem. To distinguish between them some background is necessary.

First, the general description of an optimisation problem is to find the best solution in terms of some criterion from a set of possible solutions. One of the simplest examples of this is to find the global maximum or minimum of some mathematical function $f : \mathbb{R} \to \mathbb{R}$. In that situation, a *solution* would be any real number, the *solution space* is $\mathbb{R}$, and the optimal solution is the input x for which $f(x)$ is the maximal (or minimum) value of $f(x)$.

Finding this optimal solution can be done through multiple approaches. One option is to treat the problem as a *search* for the optimal solution, with the *solution space* as the *search space*. With this approach, an option is to try every possible solution in the search space, and thereby ultimately finding the optimal one. This is called an *exhaustive* search, and relies on the fact that the search space is small enough to be searched in its entirety. If

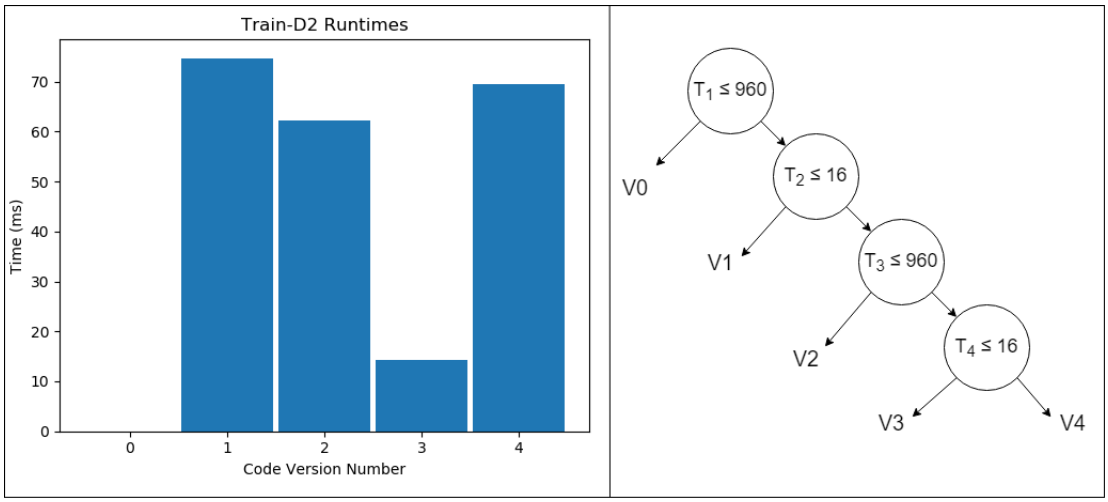


Figure 2.3: *Runtimes of the dataset Train-D2 from the SRAD benchmark, accompanied by a tree with threshold comparisons. Code version 0 is not included, due to having a runtime greater than 1 second.*

the simple case algorithm from earlier was described in this setting, it would amount to a smart exhaustive search strategy across all code versions.

An alternative is to employ some heuristic or other search algorithm, such as Gradient Descent [19]. In this algorithm, the idea is to pick a random start point for the search, and after each attempt at a solution compute the gradient with respect to the objective function to be optimised. The gradient will then influence the changes made for the next iteration, such that the gradient is followed towards a locally optimal solution. This method relies on the problem having a computable gradient, but also risks finding a local extremum and not the global extremum if the objective function is not convex. Gradient Descent is a widely studied topic, with multiple enhancements and modifications available, such as introducing a momentum term to help speed up the process [19].

Regardless of the choice of technique, it has to be applicable to the actual scenario. In the tuning setting, the only known information is the different values of $E_{par}(Input)$ for every dataset. From this, the same logic of small changes resulting in the same code versions used in the simple case still applies. For example, if a threshold is compared against $E_{par}(Input)$ values of $[10, 100, 1000]$, then any threshold value between $[1 - 10]$, $[11 - 100]$, $[101 - 1000]$, and $[1001, \infty]$ will all result in the same code path execution as any other in the same range. The difference from earlier is we only had a single range, now replaced by an arbitrary number of ranges to choose from depending on the number of iterations of the loop. Since this number can be large, getting all the information is also not feasible in most cases, meaning any algorithm also has to take into account the cost of information retrieval.

The problem space also has the additional challenge of having an objective function which isn't continuous, but rather that of a step-function due to the intervals mentioned earlier. That means that in one case, a small change to a threshold can either not change the result at all in the case that the same execution path is chosen as before, but a similarly small change might radically influence the runtime, if a better path is chosen.

Since the objective function isn't continuous it does not have a computable gradient. This is

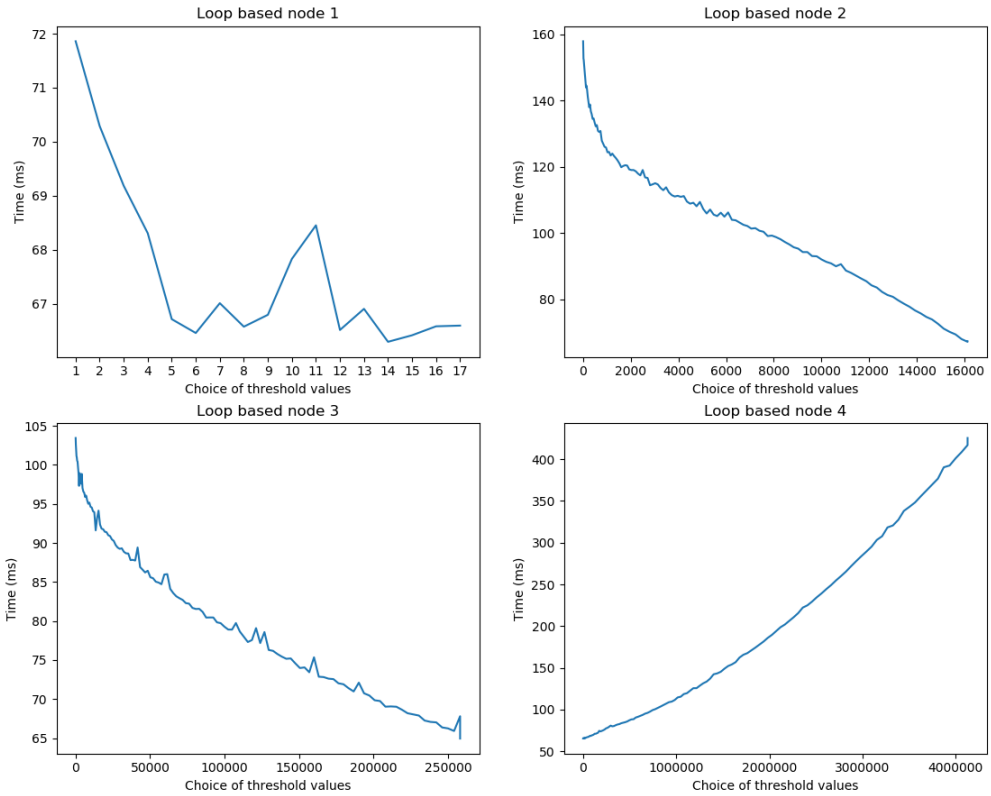


Figure 2.4: *Runtimes of a fully exhaustive run of all values of $E_{par}(Input)$ in LUD, using all datasets.*

relevant to a great many optimisation techniques, including most notably gradient descent which relies entirely on this kind of information [19]. This is also a challenge for gradient-free optimisation techniques which try to emulate a gradient, since changes in the input has inconsistent effects on the output.

To illustrate this, Figure 2.3 shows the runtime of each code version for one dataset in the SRAD rodinia benchmark belonging to the simple case. This graph also annotates the $T \leq E(Par)$ used to choose datasets. This shows how, if one first attempts a benchmark with $T_4 = 20$, and then changes that slightly to $T_4 = 25$, then no change happens in the result, since the same version was chosen. However, if we had changed to $T_4 = 15$, we would have seen a radical change as we would have gone from version 4 to version 3, which was the optimal one. This concept extends to multiple datasets, and in turn an objective function like total runtime. While the program in Figure 2.3 belongs to the simple case, the same applies in the loop based case but with more plateaus in the function.

That is not to say that some information about the underlying objective function denoting runtimes can not be gleaned from the theoretical aspects of Incremental Flattening. Using the values of $E_{par}(Input)$ as accurate expressions of the degree of parallelism used in an optimisation, it stands to reason that only one value will be optimal. The more you then deviate from that optimal value in either direction, the runtime will be negatively influenced, as either too little or too much parallelism in the hardware is being used. From this, it could be assumed that the negative impact of deviating further and further from the optimal will increase monotonically.

Under this assumption it is possible that the objective function can be roughly modelled

as a second degree polynomial. In practice this reasoning seems to hold, but the experiments do not result in proper polynomials due to the added noise and irregularities between benchmark runs. To illustrate this idea of it being close to a polynomial, Figure 2.4 shows the runtimes of every single value in the largest sequential loop program that was investigated, Lower-Upper Decomposition (LUD). The figure is created by exhaustively attempting every single uncovered value of $E_{par}(Input)$ in each loop based threshold, using all datasets available and reporting the cumulative sum of runtimes.

The figure lends credence to the idea that the underlying functions can behave much like a second degree polynomial. The exception would seem to be the first loop based node encountered, starting from the top, which has multiple values around the same runtime, along with two spikes. While this could be taken to show how this assumption doesn't fully hold, as it is still an assumption, it could simply be due to the small amount of values for $E_{par}(Input)$ in that threshold comparison, meaning what we see is simply noise due to how little the parallel size changes.

Looking at the four figures combined, it shows that for LUD the optimal value for most nodes is to simply have all thresholds be false in all iterations, except for the fourth and deepest node, which should always evaluate to true. That assignment produces the global minimum across the entire tree, and it does seem to follow the idea that the further away from the global minimum, the worse the performance gets, and that performance penalty increases monotonically. All of the above background will be relevant to all attempts to solve this problem.

Multiple approaches can have different qualities which might make them preferential in comparison to others. The following chapter will cover a few different approaches that have been attempted. All will have their algorithm explained in detail, along with theoretical reasoning as to why each could solve the problem, and experimental results for each will be presented in Chapter 4. Each of them only deal with the thresholds contained in sequential loops, and use the earlier strategy in Algorithm 2 for any simple case comparison nodes.

In order to merge them with the simple case tuner, `OptimiseProgram` from Algorithm 1 is changed with the following pseudocode from Algorithm 3.

Essentially, a quick scan of the values of $E_{par}(Input)$ each threshold T in a tree $\mathcal{T}$ are compared against is made. If all the thresholds are compared only against a singular value in every dataset, it falls under the simple case and the tree is optimised using `OptimiseSimpleTree` from Algorithm 2. If that is not the case for any threshold, one of the variants for dealing with the loop based case is used.

It should be noted that in practice trees containing difficult nodes also contain some simple case nodes. Instead of re-inventing a solution to these nodes, they are all solved by `OptimiseSimpleTrees`. For the loop based tuners, their pseudo code work on the assumption that all nodes in the trees are complicated, to avoid unnecessary clutter.<

Algorithm 3 Change to Algorithm 1 to handle loop-based trees.

Input: Program P with trees $\mathcal{T}^1, \cdots, \mathcal{T}^K$, and Datasets $D_1, \cdots D_M$.
Output: Optimal threshold assignments $(T_1^i)^*, \cdots (T_{N_i}^i)^*$ for each tree $\mathcal{T}^i$.

```
1: Trees = { }
2:
3: for all trees T^i do
4:    Loop = False
5:    for all thresholds T^j in T do
6:       Ē = GetAllValuesOfE((D1, ··· , D_M), T)
7:       if |Ē| > 1 then
8:          Loop = True
9:       end if
10:   end for
11:
12:   if Loop then
13:      Trees[i] = Execute loop-based strategy     (Algorithms 4, 5, 6 or 8)
14:   else
15:      Trees[i] = OptimiseSimpleTree(T, (D_1, ··· , D_M))    (Algorithm 2)
16:   end if
17: end for
18:
19: Return: Trees
```

CHAPTER 3

LOOP-BASED TUNING STRATEGIES

This chapter is split into five segments, each corresponding to a different tuning strategy. They are presented in the order of when they were developed during the process of writing the thesis, and will cover the reasoning for trying them, pseudocode of the strategy, and discussions of their pros and cons. They are as follows:

1. Exhaustive search, in which the idea of getting complete information is examined in the new setting.

2. Binary search, a relaxation on exhaustive search, in which less benchmark runs need to be attempted under the assumption of a polynomial and monotonicity.

3. Evolutionary strategies, a class of black box optimisers for solving continuous optimisation problems, illustrating the performance of an algorithm with no domain specific knowledge [20].

4. Active learning, a technique in machine learning whose uses fit very well into the setting of the tuner, focused on maximising the benefit of labeling unknown datapoints [21].

5. Program instrumentation, a completely separate concept in which the data accessible to the tuner is examined and optimised by modifying the target program.

For all of these tuners, they will have been tested on the dummy matrix matrix multiplication program from Listing 1.6 in Chapter 1. To recap, it is standard matrix matrix multiplication performed inside of a sequential loop, in which the sizes of the square input matrices are reduced by half in each iteration. Because of this, the parallel size involved are changed between iterations, meaning it belongs in the difficult case.

To underline the difficulty involved, for the largest testing dataset in the program, each of the four thresholds are compared against 12 distinct values of $E_{par}(Input)$. This is relatively few values since the code reduces the matrices size by half each iteration. In other cases, such as the LUD benchmark, this number reaches 255 separate distinct values.

Due to being a relatively easy instance of the difficult problem, this dummy example was used to develop the different tuners on. In the following segments, when practical results are mentioned it refers to results gained on this case, with the more difficult LUD benchmark saved for Chapter 4.

3.1 Exhaustive Search

The simplest possible solution to this problem is to try every possible value of $E_{par}(Input)$, and to choose the optimal through those experiments. This is in essence a continuation of the simple algorithm, in which all information possible is gathered, and a judgement is made based on that complete information. In terms of an optimisation technique, this is *exhaustive search*, since the search space of all possible values for a threshold is searched exhaustively. In practice, this choice of strategy has the benefit of always producing optimal results, but it has the major downside of being infeasible for larger programs to be tuned in a set amount of time. It can be implemented plainly by the following pseudo code in Algorithm 4, inserted into Algorithm 3 to finalise a tuner.

Algorithm 4 The exhaustive search optimisation strategy

Input: One tree $\mathcal{T}$ with threshold names $T_1, \cdots, T_N$ and Datasets $D_1, \cdots, D_M$
Output: Threshold assignments $T_1^*, \cdots, T_N^*$ for the N thresholds in $\mathcal{T}$

```
opt_thresholds = { }

for all thresholds T_i in DFS(T) do
    Ē = GetAllValuesOfE((D1, ··· , D_M), T) ∪ ∞
    B̄ = [ ]

    for all E_j ∈ Ē do
        B̄[j] = BenchmarkSingleChoice(T, T_i, E_j, (D_1, ··· , D_M))
    end for

    k_best = argmin(B̄)
    opt_thresholds[i] = Ē[k_best]
end for

Return: opt_thresholds
```

It's implementation is by far the simplest, shown in Algorithm 4. It simply loops over the values of $E_{par}(Input)$, attempts a benchmark per value, and chooses the one which performed the best. The resulting value is however produced from all datasets at once, instead of on a per dataset basis. Changing this is due to how ranges don't scale well into this more difficult class, since the chances of two ranges intersecting is very small. In the simple case, it was either $[1, E]$ or $[E, \infty]$ per dataset, but here the number of choices again scale with the number of values of $E_{par}(Input)$. While this change can make the individual smaller datasets perform slightly worse, as accumulative runtime prioritises larger datasets, this was left as is in the implementation.

As for the strategy's downside of long tuning time, it does turn out to be noticeable in practice. The results gathered in experiments perform very well, but the time it takes to finish tuning in some cases become fairly extreme. It is also compounded in the future as more features has to be tuned, since the search space gets exponentially bigger. For instance, some programs might require register and block tiling to get the best result, which would then require adding a tile size parameter to be tuned as well in order to get optimal performance. The choice of tile size by itself is an unsolved problem, and is mostly done by testing a few

different values chosen by heuristics. If the autotuner is to stick to the exhaustive search strategy entirely, it would in the worst case have to also redo all the work done up till that point, but with a different tile size for every possible tile size. As more tuning parameters might be added, such as workgroup sizes, the search space will continue to explode, which in turn makes an exhaustive tuner impractical to use on larger programs with additional parameters.

As it stands in the current version of what was implemented, the tile size tuning is done in only one pass through a few possible values at the end, which does not explode the search space, but makes the solution only near-optimal. Properly tuning for tile sizes is left for future work. Doing the tile size tuning in a single pass is kept for all following strategies too.

3.2 Binary Search

A variation of the exhaustive search employs a search strategy in order to not perform useless benchmarks, using some of the background knowledge. *Binary search* is done by taking the list of elements we search through, and splitting it at the middle into two segments at each iteration. If the benchmark using the midpoint is better than the previous best benchmark, the search continues using the midpoint of the segment in the direction of the best benchmark, discarding the other segment. The algorithm finishes once the segment consists of a single element. With this strategy, we gain the following guarantee:

Applying binary search always requires exactly $\log_2(N)$ *benchmark runs, with N being the number of possible* $E_{par}(Input)$ *values to search.*

This strategy has the benefit of being much faster than *exhaustive search*. It's results are however not guaranteed to be optimal, as some objective functions such as the fourth example in Figure 2.4 can have an unexpected shape. From an optimisation strategy point of view, it relies entirely on the assumption of monotonicity in the objective function covered earlier with Figure 2.4, since the binary search algorithm finds a local minimum only, and would potentially get stuck if there are multiple such local minima.

A pseudocode algorithm for this strategy can be found in Algorithm 5. This algorithm was attempted as a natural extension of exhaustive search, and also due to how it behaves much like a gradient descent algorithm, in that it follows a quasi-gradient of change to find a local minimum. Gradient descent as covered earlier can not be applied easily, as the problem isn't differentiable, but lead to the examination of binary search. Binary search as opposed to gradient descent also has a nice provable number of runs to find a local minima, whereas gradient descent does not. Ultimately, binary search shows that the concept of following a quasi-gradient can work in this setting, and that tuners can be built on the assumptions presented in Section 2.2.

Together, inserting Algorithm 5 into Algorithm 3 produces a tuner which handles all the different types of programs known, with optimal results in the simple case and near-optimal in this difficult case while maintaining reasonable tuning time

Algorithm 5 The loop-based case using a Binary Search strategy

Input: One tree $\mathcal{T}$ with threshold names $T_1, \cdots, T_N$ and Datasets $D_1, \cdots, D_M$
Output: Threshold assignments $T_1^*, \cdots, T_N^*$ for the N thresholds in $\mathcal{T}$

opt_thresholds = { }

for all thresholds T_i in `DFS`($\mathcal{T}$) **do**
  $\bar{E}$ = `GetAllValuesOfE`($(D1, \cdots, D_M), T) \cup \infty$

  $Lower = 0$
  $Upper = |\bar{E}| - 1$

  $Time_{best}$ = `BenchmarkSingleChoice`($\mathcal{T}$, T_i, $\bar{E}[Upper]$, $(D_1, \cdots, D_M)$)
  $Index_{best} = Upper$

  while $Upper - Lower > 0$ **do**
    $Midpoint = \lceil (Upper + Lower)/2 \rceil$
    $Time_{new}$ = `BenchmarkSingleChoice`($\mathcal{T}$, T_i, $\bar{E}[Midpoint]$, $(D_1, \cdots, D_M)$)

    if $Time_{new} < Time_{best}$ **then**
      $Time_{best} = Time_{new}$
      $Index_{best} = Midpoint$
      $Lower = Midpoint + 1$
    else
      $Upper = Midpoint - 1$
    end if
  end while

  opt_thresholds[i] = $\bar{E}[Index_{best}]$
end for

Return: opt_thresholds

3.3 Evolution Strategies

One candidate black box optimisation algorithm which has been investigated is the continuous optimisers in the evolutionary strategies (ES) family. Black box optimisers make no assumptions on the problem at hand, with ES being inspired by Darwin's theory of evolution to guide the search for good solutions [22]. In this section a popular implementation of ES called CMA-ES will be applied to our tuning problem.

Theory of Evolution Strategies

ES are useful optimisation strategies in situations where a large solution space has to be searched efficiently, with multiple dimensions in the input. A basic instance of an ES works by creating a *population* of candidate solutions at random, called *individuals*, and by evaluating them on the problem. Each individual's *fitness* is measured to gauge how well it solves the problem. From that fitness only the fittest individuals survive through a *selection* process, such as an *elitist* one where only the most fit survive. These surviving individuals then spawn offspring, which are imperfect copies of their *parent*, meaning they have mutated

slightly. These changes can either be from *crossover*, which is the mixing of two parents genomes together, or some other random *mutation*. These offspring and parents are then reentered into a tournament for new survivors generating new offspring, until the problem is solved. This structure is illustrated in Figure 3.1.

The core idea of ES optimisation is that of a random process, with a probability distribution generating offspring guided by the objective function. This has the advantage of not needing a gradient, but with the drawback in cases such as step functions where the quasi-gradient guiding the evolution might mislead it. This can be alleviated depending on the implementation of the mutations of the individuals. For instance, an example of a mutation in our setting could change an individual from one code version to another, since the cutoff points are known in advance. Another mutation could set a random threshold to entirely true or false, since that helps give information about removing the influence of a specific threshold.

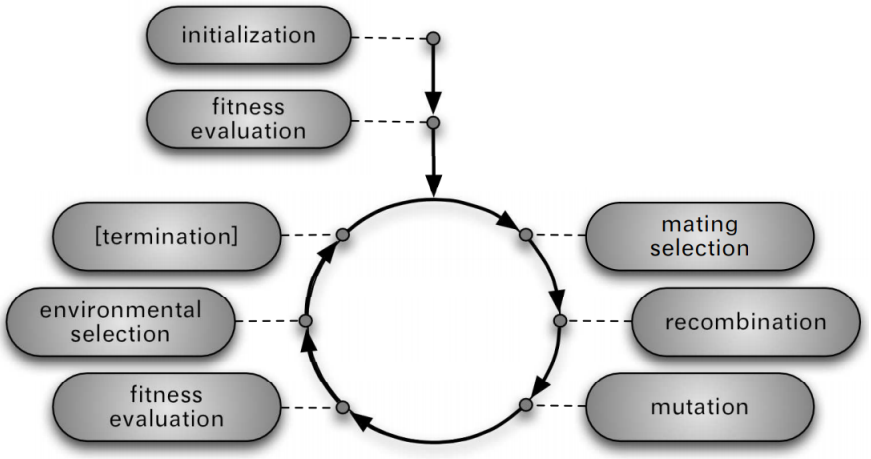


Figure 3.1: *Typical steps of a genetic algorithm. Mating selection, recombination and mutation generates a pool of new candidates, which are tested and only the best survive for the next iteration. Source: Igel [23]*

As mentioned, we will specifically apply CMA-ES to the tuning problem. The CMA in CMA-ES stands for covariance matrix adaptation, which was introduced in the context of ES in order to formulate the notion that recent successively selected mutations continues to be beneficial, thereby improving convergence of ESs [24]. More specifically, the mutations of an ES are drawn from a distribution which is typically initialised as a normal distribution. This distribution is then *adapted* to the environment in the form of the specific problem instance. This adaptation is done using the covariance matrix, and the CMA algorithm updates the matrix such that the probability of recently beneficial steps in the evolution are more likely in the near future. CMA ensures that this adaptation to the environment is efficient, and converges faster, as described in [24] and [25].

Converting Tuning to an ES problem

As mentioned before, ES are black box optimisation algorithms which work well for a wide range of problems with large search spaces. They require very little knowledge of the problem, except how solutions are expressed, and how they are individuals are evaluated. Applying the CMA-ES implementation used [26] to our problem only requires a sample solution encoded as a vector of real numbers, and a fitness function for evaluating solutions of the given

form, leaving the mutations and adaptation to the algorithm itself.

In the case of tuning, the simplest formulation would then be to plainly supply a valid threshold configuration randomly sampled from a normal distribution, and have the fitness function simply be accumulated runtime across all datasets. This threshold configuration could then either correspond to tuning a single threshold, or alternatively all thresholds in a single tree. While the other tuners that were and will be presented have all focused on a singular threshold at a time, ES in general are efficient at problems with multiple dimensions and large search spaces. Tuning the entire tree in one application of the algorithm is part of the reason this style of tuner has been investigated.

Besides the initial solution and fitness function, a few additional hyperparameters for a CMA-ES application can be tuned separately, with the two most significant being the initial variance σ and population size. Variance refers to the spread of possible solutions in the search space of solutions, with the default value suggested by a popular python package implementation of CMA at 1/4th of the search space width [26]. For instance, if we know an optimal threshold lies between 0 and 100, we could provide a sample solution of 50 with a σ of 25. As the CMA-ES begins newer iterations, this σ is changed to better adapt to the problem environment.

Population size is not an initial size, instead denoting the number of solutions to attempt in each iteration, before moving onto the next generation of solutions. The same package as mentioned earlier suggests a default value of $4 + 3 \cdot log(N)$ where N is the number of dimensions in the search space [26]. For the tuning problem, this is the number of thresholds in a tree.

The major reason σ and population size are of interest to tune is they allow for specifying the amount of exploration the ES does of the search space. Increasing the variance allows for a larger initial spread of solutions, while a larger population size increases the chances of that exploration finding a better solution to select for future iterations. Increasing either comes at the cost of increased tuning time, which again is relevant to our problem.

Practical Problems

The major problem that was encountered during development of this tuner has been the effect of step functions on CMA-ES. As mentioned previously in Figure 2.3, the objective function is in fact a step function, meaning CMA-ES is not entirely suitable for this project as it is for continuous optimisation. With caching implemented using the naïve initial solution from earlier, each new generation with ten iterations could have only two of them find an unexplored plateau in the step function, and the rest would have the same execution path as an earlier attempt, but with a slightly different configuration. When this happens for the majority of a population in a generation, the algorithm might falsely assume convergence, and thus terminate without finding the correct minimum.

While this is not ideal, one way of dealing with this is to increase the σ beyond the default value, yielding a greater probability of individual solutions reaching different plateaus, and likewise for population size. Ideally this problem is handled by a reformulation of the problem conversion, specifically the type of solutions produced by the ES. As mentioned before, these are based on the initial solution we supply, but the naïve conversion shown

above leaves open the problem of multiple different solutions being equivalent.

One way of solving this is to count the number of unique threshold configurations possible in the tree, and have CMA-ES choose one of these. For the ES to properly choose from them, it still has to retain the same information about how changing one threshold influences the runtimes.

In the tuner implementation this is done by finding a list of all $E_{par}(Input)$ values pr. threshold in the tree, and having CMA-ES find indices in that list. The change means that the solution space to be searched becomes much smaller, but at the cost of some information of the scale between the ranges. As an example, if a threshold encounters the values $(1, 5, 10, 1000)$ for $E_{par}(Input)$, the ES would generate an integer solution corresponding to an index into said list, which means it can choose between $(0, 1, 2, 3)$. The search space has been shrunk from 1000 possible values to just four, retaining most of the relevant information.

Using this strategy was efficient in reducing individual solutions being repeats of earlier ones. Experimenting with the LUD benchmark introduced earlier, the naïve encoding would have a repeat percentage of 95%, meaning very few meaningful experiments were attempted. Alternatively, encoding with indices brought said percentage down to 53%. For later experimental comparisons, the index encoding will be used.

As an additional enhancement, the initial solution supplied was also changed to being a solution where every threshold is false. Recalling the earlier tuners, they all used the fully flattened version as a good starting point, and this should extend to CMA-ES. In CMA-ES, this should help avoid a node near the root being set to true, thus invalidating any changes to thresholds deeper in the tree.

Downsides and Conclusion

Being a random process should take its toll on tuning time for the difficult case. Unlike the binary search tuner, no guarantee for the number of benchmarking runs can be made, but it is possible for it to randomly find the optimal early on.

As for the quality of the results, the tuner should produce decent results comparable in quality to that of the binary search, but with the caveat of it taking longer, and in some instances finding bad solutions. In practice this tuner is impractical, but was explored as an example of applying a black box optimiser to the problem. It shows that while black box optimisers can provide powerful tools, it is not every problem which can be solved using them.

3.4 Active Learning

While CMA-ES and black box techniques were attempted to little success, other aspects of machine learning are still relevant to investigate. *Active learning* is a general technique in machine learning where the model being trained is an active participant in its learning process, meaning it is not an algorithm by itself, but rather a technique used to augment training in a specific setting. This technique applies in settings with a set of unlabeled datapoints, but where the process of labeling these is expensive and left to the model being trained. The goal of active learning is to optimise the value of each labeling, such as to

maximise the information gained about the unknown objective function, and thereby the learning of our model. This is opposed to standardised batch learning, in which all data is present to start.

Considering the setting of the tuning problem, incorporating active learning makes sense, since we have a set of unlabeled data points we want to predict in the form of possible threshold configurations, but to benchmark them is the most costly expense in the process. Active learning aims to accurately predict the objective function, while minimizing the number of benchmarks needed to model it well enough to find the minimum. It is also fitting, since we do have some assumptions about how to model the underlying function, using a second degree polynomial as in the binary tuner.

To illustrate the general idea, Burr Settles [21] created a toy example shown in Figure 3.2 underlining the benefit of active learning. In his example, 400 samples were drawn from two normal distributions with associated label for the distribution which generated the sample. If these 400 samples were given as unlabeled data, with the goal of accurately labeling all 400 in only 30 samples, the figure shows the difference between learning the same logistic regression model on 30 randomly chosen samples versus 30 samples chosen by active learning. In his example, the difference in accuracy was 0.7 for the random samples, and 0.9 for the active learning samples.

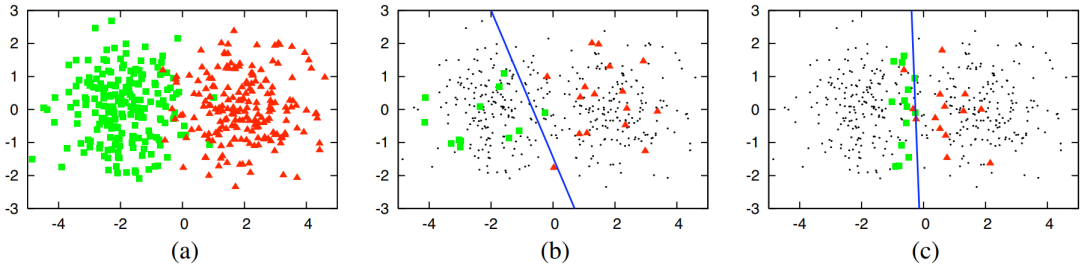


Figure 3.2: *(a) 400 instances sampled from two different normal distributions. (b) Decision boundary of a logistic regression trained on 30 randomly sampled points. (c) Decision boundary of a logsitic regression trained on 30 samples chosen by active learning*
Source: Burr [21]

In the same way that evolutionary strategy applications differed from each other in their mutations, so does active learning in its *information metric* for choosing the most optimal unlabeled datapoint to query. Multiple approaches exist, such as choosing the points which the model is most uncertain about, the points which we expect might most change our model, or any other situation specific heuristic. While most of these metrics focus on classification rather than regression [21], the primary information metric we have experimented with is to use the assumption of a polynomial objective function to query the predicted minimum value in the unlabeled data.

To illustrate that specific implementation, pseudocode for it is provided in Algorithm 6, with convergence being determined when the optimal choice doesn't change between iterations. For every threshold T_i in tree $\mathcal{T}$, three initial randomly sampled guesses for $E_{par}(Input)$ values are benchmarked. A second degree polynomial model is then fitted using those labeled datapoints, and all unlabeled datasets are predicted using that model. In every iteration of

Algorithm 6 The loop-based case using an active learning strategy

Input: One tree $\mathcal{T}$ with threshold names $T_1, \cdots, T_N$ and Datasets $D_1, \cdots, D_M$
Output: Threshold assignments $T_1^*, \cdots, T_N^*$ for the N thresholds in $\mathcal{T}$

1: opt_thresholds = { }
2:
3: **for all** thresholds T_i in `DFS`($\mathcal{T}$) **do**
4: $\bar{E}$ = `GetAllValuesOfE`($(D1, \cdots, D_M), T$) $\cup$ ∞
5:
6: *Guesses = Randomly sample 3 values from* $\bar{E}$
7: $\bar{E}_{unlabeled} = \bar{E} \cap Guesses$
8: $\bar{E}_{labeled} = \emptyset$
9:
10: **for all** $e_{guess} \in Guesses$ **do**
11: $Time_{guess}$ = `BenchmarkSingleChoice`($\mathcal{T}$, T_i, e_{guess}, $(D_1, \cdots, D_M)$)
12: $\bar{E}_{labeled} = \bar{E}_{labeled} \cup (e_{guess}, Time_{guess})$
13: **end for**
14:
15: Converged = False
16: **while** *Not Converged* **do**
17: *Model = Second Degree Polynomial Linear Regression of* $\bar{E}_{labeled}$
18:
19: $E_{predicted} = Model(e)$ *for* $e \in E_{unlabeled}$
20: $e_{guess} = argmin(E_{predicted})$
21:
22: $Time_{guess}$ = `BenchmarkSingleChoice`($\mathcal{T}$, T_i, e_{guess}, $(D_1, \cdots, D_M)$)
23: $\bar{E}_{labeled} = \bar{E}_{labeled} \cup (e_{guess}, Time_{guess})$
24:
25: *Converged = True if same "predicted guess" twice, i.e. no change in model.*
26: **end while**
27:
28: $Index_{best} = argmin(\bar{E}_{labeled})$
29: opt_thresholds[i] = $\bar{E}[Index_{best}]$
30: **end for**
31: **Return:** opt_thresholds

a while loop until convergence, the best predicted unlabeled datapoint is labeled, and the second degree model trained again on the updated labeled datapoints. When convergence is reached, the optimal threshold is chosen amongst the labeled datapoints. The overall structure of the pseudocode remains the same for any other choice of information metric and convergence condition.

Different parameter choices, implementation variations, and additional improvements have been experimented with. The pseudocode above prioritises fast prediction, as opposed to certainty about the quality of its result, but serves as a base for expansion. Like the binary tuner, it works solely on the assumption of the objective function being reflected by a second degree polynomial, as roughly illustrated by Figure 2.4. Here the assumption serves as the base for predictions, as seen in Lines 17 - 20.

The simplest parameter to tune would be the number of initial guesses, which was kept as three samples. We observed that when this was increased to ten, the quality of solutions did not increase on average across five runs. Usually, the added random guesses did not provide enough information to the model to outperform the guesses that instead would be generated by the information heuristic. Instead of adding more initial guesses to get better quality results, changing convergence condition helped improve accuracy.

To underline why changing convergence is necessary, consider the list of datapoints [5,4,6,7,8] where we want to find the minimum value, but they start as unlabeled. If we were to follow Algorithm 6 and sample three points, we could end up with $[\_, \_6, 7, 8]$, with _ denoting unlabeled points. The second degree polynomial fitted to the three known points would indicate a clear minimum at index 0, and attempt that value giving $[5, \_, 6, 7, 8]$. Training the model again would not indicate in any way that the final unlabeled datapoint would be better, and thus not try it. This is an example of the naïve convergence condition in Algorithm 6 being inadequate.

Endpoint Sampling

The alternative convergence condition we decided on using to correct this behaviour and ensure more confidence in the result, was to do *endpoint sampling*. The idea is that the guess produced by the above algorithm is at least close to the optimal, but sampling points in the vicinity of that guess should either confirm or deny whether the predicted optimal is indeed correct. If no point in the vicinity beats the time of the optimal, then it should ideally be the optimal. Noise in the benchmark runs might negatively influence this, but overall this should add more confidence in the final choice.

In order to augment the code in Algorithm 6 with endpoint sampling, a few modifications are made. First, Line 23 is changed to keep track of any change to the best runtime in $\bar{E}_{labeled}$, rather than in the predicted optimal. This is done since we no longer trust the prediction as being correct. In addition, after the check for convergence in the old algorithm, endpoint sampling is introduced, testing the neighbouring values of the predicted best value. If either neighbour beat the previous best experimental result, we reattempt the loop, and if neither beat it then we have converged. This can be seen in algorithm 7, which is to replace Lines 16 - 26 in algorithm 6.

Adding this change provides an active tuner prioritising good results, rather than speed, as the convergence only stops when a proper minimum is found. This can be further improved by correcting for noise, by also trying the neighbours even further away, but in essence it is the same as the above. Using this strategy we found that three initial guesses still worked the best, as the primary driver for finding the minimum becomes the convergence criteria, with the initial guesses simply pointing to the local area in which the optimum is believed to be.

In experimentation this style of tuner performs better than the evolutionary strategy tuner in quality and speed, but performing only about as well in both speed and quality as the binary tuner. The specific experimentation results are kept for the following chapter, but in general this tuner shows some promise when looking for fast near-optimal results, but with the option of tuning parameters to allow for higher confidence at the sacrifice of speed.

Algorithm 7 Endpoint sampling augmentation of the Active Tuner

while *Not Converged* **do**
  $e_{best} = min(\bar{E}_{labeled})$
  $Model = Second\ Degree\ Polynomial\ Regression\ of\ \bar{E}_{labeled}$
  $E_{predicted} = Model(e)\ for\ e \in E_{unlabeled}$
  $e_{guess} = argmin(E_{predicted})$
  $Time_{guess} = \texttt{BenchmarkSingleChoice}(\mathcal{T},\ T_i,\ e_{guess},\ (D_1, \cdots, D_M))$
  $\bar{E}_{labeled} = \bar{E}_{labeled} \cup (e_{guess}, Time_{guess})$
  if $(e_{best}, Time_{best}) == min(\bar{E}_{labeled})$ **then**
    $Index_{best} = argmin(\bar{E}_{labeled})$
    for all $e_{endpoint} \in [\ \bar{E}[Index_{best} + 1],\ \bar{E}[Index_{best} - 1]\ ]$ **do**
      $Time_{endpoint} = \texttt{BenchmarkSingleChoice}(\mathcal{T},\ T_i,\ e_{endpoint},\ (D_1, \cdots, D_M))$
      if $Time_{endpoint} < Time_{best}$ **then**
        $Converged = False$
        break
      else
        $Converged = True$
      end if
    end for
  end if
end while

3.5 Instrumentation Tuner

During the process of developing the previous loop-based solutions, a more efficient theoretical solution was discussed, but which required more compiler-level implementation to achieve. The earlier methods for the loop-based case all uses the Futhark benchmark tool post compilation, and work solely on the data available in the runtime of an entire program execution. In terms of loop iterations, this data contains the summation of all loop iterations, which so far has not been sufficient to provide fast estimation of the underlying objective function for unknown labels for any of the earlier algorithms.

This inefficiency led to investigating how to obtain data that was more indicative of the underlying characteristics of the code versions. Program instrumentation is a technique of modifying existing programs, such as inserting additional code at specific points in the control flow. Using this technique, the idea is to substitute the data from a summation of loop iteration runtimes to a list of individual loop iteration runtimes. This new data will be shown to be much more useful at gaining information about the system, leading to an algorithm which in theory produces optimal results, requiring only one benchmarks per code version in the combined program.

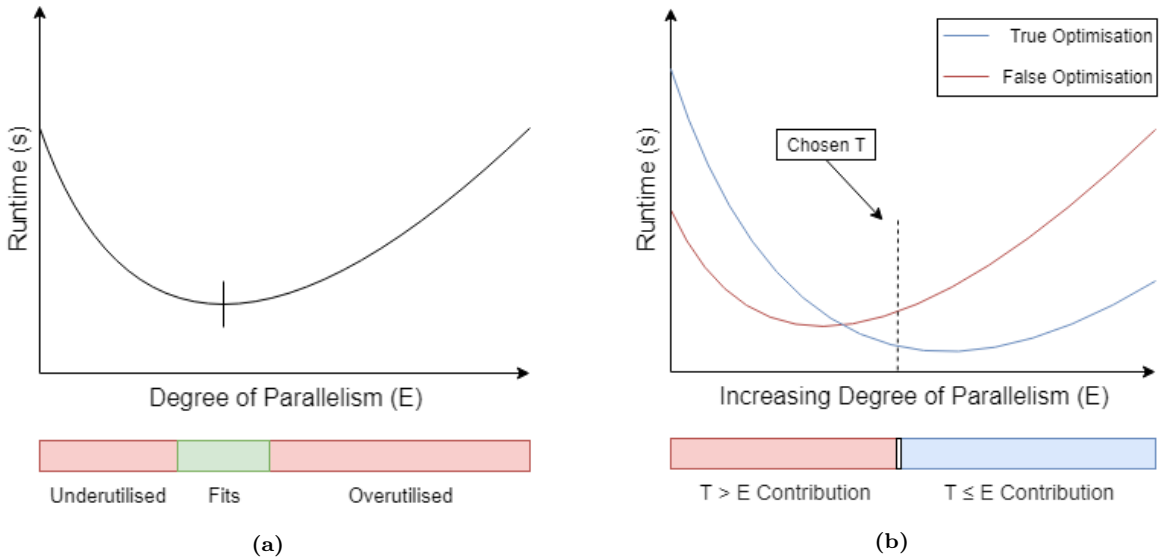


Figure 3.3: *(a) Illustration of the relationship between input dataset's parallelism requirements and the amount available in hardware under a single optimisation. (b) Visualisation of the contribution of each choice of code version optimisations in a comparison node, with a chosen threshold value T.*

Change of Data

In order to frame the importance of the change of data, recall the abstracted goal of the tuner. The goal as stated previously is to find the thresholds for which the correct of two optimisations are chosen, depending on dataset and hardware characteristics. The behaviour of a single such optimisation when plotting the degree of parallelism of the input dataset against the runtimes was in the case of Binary Search and Active Tuning assumed to be polynomial in nature, as per Figure 2.4. The minimum of that plot would be the point in which the degree of parallelism supplied by the dataset fits perfectly onto the GPU hardware. Any dataset with a lower degree of parallelism would underutilise the hardware, and conversely if it was greater it would require more of the hardware than it can deliver, both of which leading to significantly worse performance for that optimisation [12]. One example of the latter is for Block Tiling to optimise spatial and temporal locality using shared memory. Implementing block tiling to take advantage of fast shared memory is very efficient, but if the data would require more shared memory than is available the optimisation's performance suffers. This behaviour is sketched in Figure 3.3 (a).

Returning to the goal of the tuner, the goal is to choose the threshold closely representing the degree of parallelism for which switching from the false optimisation to the true optimisation minimises the total runtime over all the iterations. Thus, the goal is not to find the degree of parallelism minimising the runtime of one optimisation, but rather optimising the cumulative runtime when combining two such curves.

Consider the situation in Figure 3.3 (b), in which the degree of parallelism steadily increases with each iteration. The two curves correspond to the two code versions chosen between in a comparison node with a chosen threshold value T. As long as the value of E is less than T, the predicate $T \leq E$ evaluates to false, and only the red curve contributes to the overall runtime. When the values of E are large enough that the predicate becomes true, the only contribution to runtime comes from the blue curve.

This way of looking at the relationship between the optimisation graphs and runtime becomes what will be called the *combined area under the curves*. Finding the optimal threshold with these two curves now become the task of finding the threshold which closely matches the degree of parallelism for which the combined area under the curves is minimised. This corresponds to the following formula:

$$\text{Optimal Threshold } = \text{argmin}_T \left(\sum_{i=0}^{T} OptFalse[i] + \sum_{i=T}^{\infty} OptTrue[i] \right) \tag{3.1}$$

In formula 3.1, $OptTrue$ and $OptFalse$ refers to a list of runtime per iteration using that threshold in their respective curve, sorted by degree of parallelism. While this might be difficult to grasp to start, the following graph in Figure 3.4 from a Futhark example should show how this concept looks visually in a real example, with two optimisation curves overlapping in the same plot. In the figure, a single threshold is examined for which two code versions are being distinguished between as in Figure 3.3 (b). Looking at the graph should make it clear how at $E_{par}(Input) = [256, 1024]$ there comes a point at which switching from one code version to the other becomes beneficial, and continues to do so from that point on.

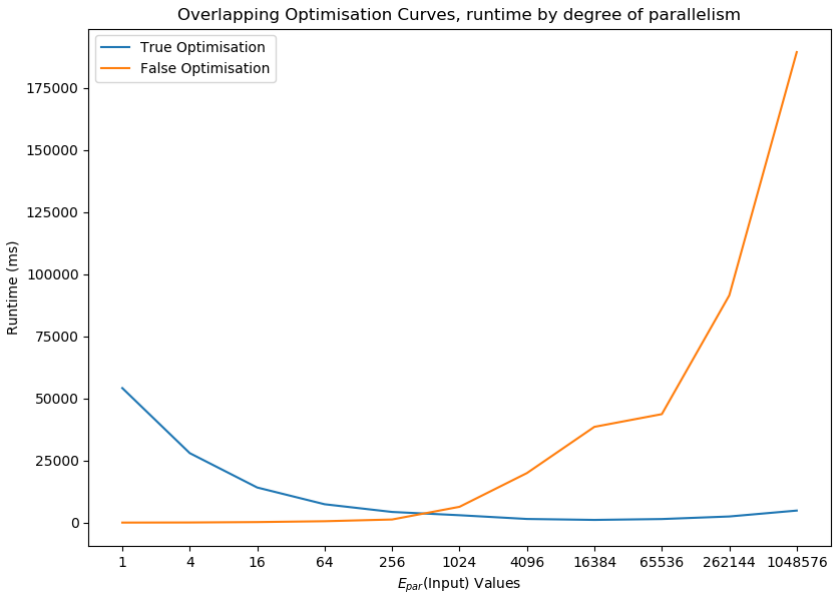


Figure 3.4: *Example of two optimisation's overlapping curves, illustrating the intuition of using the combined area under the curves as a metric for finding optimal thresholds. The two optimisations occur in a real tuning run of a futhark example, with a loop-based tree.*

Instrumentation Algorithm

With the underlying metric explained, we will present the tuning algorithm itself. Like all the other tuners, this one is recursive in nature, tuning the tree using depth first traversal, with pseudocode in Algorithm 8. One thing to note is that in the algorithm it is shown that one benchmark run is enough to give information about one optimisation curve, which will be explained afterwards when discussing the data. For the pseudocode, $\texttt{Benchmark}_{instrumented}$ returns a list of loop iteration runtimes with corresponding $E_{par}(Input)$ values. In terms of the earlier motivation, the algorithm simply computes the two optimisations' curves as in Figure 3.4, and then uses Formula 3.1 to compute the optimal threshold to choose.

In terms of tuning time, the algorithm would require one benchmark run for each code version, since no other type of run would give more information about the system. The reason the two benchmarks uses the threshold set to 1 and ∞ is to isolate the run to only choose one code version, meaning we get that code version's performance in each loop iteration, allowing us to compute the graph. An essential enhancement to the tuner is to have caching of results, since the benchmark with T set to ∞ is a combination of the two prior benchmarks of the tuning process for the node one level deeper in the tree, which is how the tuner achieves one benchmark per code version. This tuner has some striking similarities with the original simple case tuner in Algorithm 2, being that they both require the same number of benchmark runs, uses the same way of testing each code version separately, both use caching in the same way, and they both exploit *complete information* about the system to choose the threshold.

Algorithm 8 Instrumentation Tuner for Loop-based Trees

Input: One tree $\mathcal{T}$ with threshold names $T_1, \cdots, T_N$ and Datasets $D_1, \cdots, D_M$
Output: Threshold assignments $T_1^*, \cdots, T_N^*$ for the N thresholds in $\mathcal{T}$

1: opt_thresholds = { }
2:
3: **for all** thresholds T_i in $\texttt{DFS}(\mathcal{T})$ **do**
4: $(\bar{E},\ Times_T) = \texttt{Benchmark}_{instrumented}(\mathcal{T},\ T_i,\ 1,\ (D_1, \cdots, D_M))$
5: $(\bar{E},\ Times_F) = \texttt{Benchmark}_{instrumented}(\mathcal{T},\ T_i,\ \infty,\ (D_1, \cdots, D_M))$
6:
7: $Runtimes = [\ ]$
8: **for all** j in $(1, \cdots, |\bar{E}|)$ **do**
9: $Runtimes[j] = \left(\sum_{k=0}^{j} Times_T[k] + \sum_{k=T}^{\infty} Times_F[k]\right)$
10: **end for**
11:
12: $Breakpoint = \text{argmin}(Runtimes)$
13:
14: opt_thresholds$[i] = \bar{E}[Breakpoint]$
15: **end for**
16: **Return:** opt_thresholds

Complete Information of the System

The greatest strength of this tuner lies in the previous statement that this tuner gains complete information about the system it tries to model. Knowing the runtime of each optimisation at any given iteration of the loop means there is no more information any benchmark run can give us. As an example, if after performing the two benchmark runs specified in Algorithm 8 one would like to execute a benchmark run with some value of T between 1 and ∞. The total runtime of that execution could be computed entirely from the information already gained by running with T set to 1 and ∞, using the formula for the combined area under the curve in Equation 3.1. This is in essence also what is being done in line 7 of the algorithm, in which all total runtimes for all possible thresholds are computed using simple sums, after which the best one is chosen. Having the property of complete information means the resulting thresholds can be guaranteed to be optimal, assuming the runtimes of each iteration are not too noisy, much like for the original simple case algorithm.

This is in contrast to the previous loop-based tuners who all estimated the thresholds fairly well, but where the quality of the data they could extract from the system never allowed complete information to be a viable option. In order for the earlier tuners to gain complete information, they would have to perform a benchmark run for each value of $E_{par}(Input)$, for each threshold comparison in the tree, which when discussing the exhaustive search strategy was deemed infeasible.

From a data science perspective, this difference in quality of data is what it means for data to be representative of what the algorithms are trying to model. What all these tuners aim to model is the behaviour of the underlying optimisations, and not just the total runtime of the program. Since thresholds themselves are supposed to correspond to a degree of parallelism, having data only represent total runtimes was not sufficient, while data about the relation between total runtimes and degree of parallelism proves extremely valuable.

Prerequisites and Implementation Details

As mentioned earlier, this type of tuning requires a bit more work done, especially on the compiler level. The minimum information necessary for Algorithm 8 to work is for the benchmarking tool to report the $E_{par}(Input)$ for each iteration, alongside the accurate runtimes of that iteration. The simplest way to achieve this is to modify the compiler generating the Incremental Flattening code to also include timing code at the start and end of each code version, alongside the $E_{par}(Input)$. In terms of the Incremental Flattening inference rules [10] also shown earlier in Figure 1.5, this would mean a slight modification to rule G3. While this should be fairly easy to do, it was not done entirely for this thesis, meaning an alternative was used.

The plots seen in figure 3.4 were generated for runtimes in each iteration correctly, but not using proper implementation of the above strategy. Instead, the Futhark Incremental Flattening code generated includes optional debugging information which can be saved to a JSON file as plain text from `stderr` describing the execution flow of the program, by passing the program the flag '`-D`' [27]. Reading this debug information reveals memory allocation, individual OpenCL kernel runtimes, threshold comparisons and other information as a log of events during program execution. To work with this, a simple regex parser was made to identify relevant events in the log, those being kernel runtimes and threshold comparisons. Sequential loop iterations were inferred as having a repeat threshold comparison in the log, since after a threshold comparison no other comparisons against the same threshold is made until the next loop iteration.

While this debugging information was not specifically created to allow for this type of tuning, it does include the two vital pieces of information necessary. It does come with a slight caveat, since the debugging code modifies the code slightly in order to perform proper timing of individual kernels by introducing barriers. Barriers in GPU code block thread execution past the barrier, until all threads reach the same point in the code [28]. This is introduced in order to properly time the individual kernels, but it has the side effect of slowing down the execution of the entire program arbitrarily [27]. As a consequence, code versions with multiple kernels will be slowed down slightly, as some threads will be blocked from executing. This slowdown might not be consistent, meaning there is some added noise to the results, making the tuning implementation presented here a proof of concept rather than a full implementation.

Benefits and problems

In theory this type of tuner produces optimal results consistently, with a provable number of benchmarks required. Those qualities alone are exceptional in the loop-based case, but it also allows for additional flexibility in the way tuning is performed since everything could potentially be moved over to compiler level tuning. To elaborate, consider the only noticeable flaw in the current proof of concept implementation.

That downside of the implemented tuner is that to gain complete information, the tuner does have to run very sub-optimal benchmark runs as well. From experimentation, the optimal choice of optimisation is normally found relatively early on in the tuning process, meaning at a deep node in the tree. After finding this, the following experiments tuning closer to the root of the tree usually require running an extremely slow code version. While this hasn't been a major concern for the test programs I've used for experimentation, it could be a serious issue for tuning time of larger programs with many datasets.

One way of dealing with this is to further explore the possibilities of doing program instrumentation, in order to for instance package tuning inside of the compiled program. The idea would be at compilation to generate a standalone program to do the autotuning, and then produce a program with the thresholds already coded for the comparisons. Doing this could allow more radical modifications to the execution flow during tuning, such as having more accurate timeouts to cut off benchmarks that run longer than the current best, and to detect that the optimal has been found and cut off before running too many costly and unnecessary experiments. Instrumentation allows for more flexibility, such as running specific pieces of the incremental flattening code in any order, but it would also require expertise in compiler development to incorporate properly, and as such is left as future work.

As it stands, this tuner serves as the main contribution of this thesis, providing nice guarantees in all types of problems presented in tuning Futhark incremental flattening programs. The following chapter will explore the benefit of all tuners together, discussing whether the theoretical benefits of this tuner translates to experimental results.

CHAPTER 4

BENCHMARKS

In order to verify the effectiveness of the autotuning solutions, different benchmarks commonly used for parallel program performance benchmarking have been prepared to test on, to show the real world usecases of tuned incremental flattening. This chapter will first discuss the general methodology for estimating the efficiency of the tuners, including a description of the benchmark programs used, and how they are used in the context of tuner validation. This is then first applied to the benchmarks which belong to the simple case solved by Algorithm 2, followed by the difficult case using Algorithms 5, 6, 8 and CMA-ES. Discussion of the results is saved for the final chapter, with this being focused on the methodology and experiments themselves.

4.1 Tuner validation methodology

The methodology for testing both the simple and difficult case is identical, with different target benchmark programs. First, benchmarks are split into two groups, with one being the training programs used during development, and the other being kept for validation. Each of these benchmarks also needs input dataset representative of different degrees of parallelism, and finally verification of their results. This section covers all of these parts, starting with a focus on the simple case. In total we have examined 11 different real-world benchmarks, along with one dummy example crafted for the difficult case.

4.1.1 Simple Case Benchmark Programs

To refresh, the simple case refers to programs whose incremental flattening predicates only ever compare a threshold against a singular value of $E_{par}(Input)$. Most programs tend to fall under this situation, and this will also be expressed in the experimental benchmark programs.

For the first set of benchmark programs used in development, three benchmarks have been chosen in the simple case. `SRAD` from Rodinia was chosen as an example of the simplest case, with no complications to it's tree structure. `LocVolCalib`, which is taken from real-world stochastic volatility calibration in financial transactions was chosen due to having a structure with multiple optimisations on the same depth of nested parallelism. Finally we chose `BFast`, which is an algorithm for detecting changes in satelite images such as deforestation of the rainforests, since it is a large program with many separate kernels with a reference handtuned threshold configuration available.

For the validation benchmarks, a collection of five other rodinia benchmarks and two FinPar benchmarks for which Futhark implementations already exist as part of the language's benchmarking suite is chosen. Rodinia is a benchmark suite for heterogeneous computing in which each benchmark exhibits a different kind of parallelisation strategy on multi-core systems [29]. FinPar on the other hand is developed in part by the Futhark team, and contains three benchmarks which by design lends themselves to incremental flattening, as FinPar focuses on the trade-offs involved in exploiting different degrees of parallelism in datasets, which other benchmarking suites such as Rodinia usually do not consider. Here these will be presented, along with a brief description of each:

1. `Backprop`, parallelised back propagation used to train neural networks from machine learning. Neural networks contains a large amount of weights that need to be adjusted during training to minimise an error with respect to some objective function. Back propagation is the method used for computing each individual weights contribution to the overall error. This contribution relies on other weights contribution, meaning there is dependencies between each step of the algorithm [29].

2. `LavaMD`, a physics simulation in which multiple particles in a large 3D space exert forces on each other, with the simulation computing particle potential and relocation due to these forces. The parallel nature of the simulation comes from splitting the 3D space into smaller cubes, which are each delegated to a workgroup in OpenCL, but requiring communication between particles from across workgroups.

3. `NN`, the k-nearest neighbour algorithm, again from machine learning. Given a dataset of points with associated label, NN can be both a regression or classification algorithm, in which a new datapoint's value is determined by it's k nearest labeled datapoints. Since this value is chosen based on distance, it lends itself well to parallelisation as the distance to each datapoint can be computed independently [29].

4. `NW`, Needleman-Wunsch optimisation algorithm for DNA sequence alignment. The goal is to score a series of different parings of DNA sequences, and finding the most optimal based on said scores. Part of these pairings do depend on each other, but a specific parallel dependency pattern can be found [29].

5. `Pathfinder`, an algorithm relying on dynamic programming to efficiently find a minimum cost path through a maze, represented by arrays of costs. Starting from the bottom most row of a matrix, the goal is to reach the top most row, with a move being either straight ahead or diagonal. Dynamic programming involves storing solutions to subproblems, which has to be exploited in a parallelised version.

6. `Heston`, a financial sector application from the FinPar benchmark suite [30]. The Heston model is used to calibrate volatile prices of options, with the goal of appraising future options' payoff. FinPar is a benchmark suite created to illustrate GPGPU effectiveness for parallelising financial applications, with `Heston` being one such benchmark comprised of parallel constructs using different levels of hardware parallelism.

7. `OptionPricing` also from the FinPar suite, this algorithm appraises "*option contracts*" which is a common form of contracts used by financial actors. The value of such contracts lies in the future, but the `OptionPricing` algorithm for pricing them is formulated in terms of data-parallel functional constructs, much like `Heston` [30].

All the benchmarks are compiled using the latest version of Futhark as of the time of writing, being `Futhark 0.12.0`. There is one exception which is in the case of `bfast`, which is compiled using `Futhark 0.10.1`. The reason for this exception is that it was used during development of the simple tuner when the latest version was `0.10.1`, but the newer version of Futhark's incremental flattening does not produce the same code as back then. This is a problem, as the newer version has a few code versions for which the program will crash, before getting the information necessary to properly tune. In order to properly compare to the handwritten solution, the `bfast` program was compiled this way. Having this exception should not impact results, as the older version simply represents a more stable version of `bfast`, which uses a lot of `intrinsic` functions to guide code generation.

Returning to the benchmarks themselves, `OptionPricing` has the most complicated tree structure of the benchmarks, shown in Figure 4.1. This is the only benchmark where a threshold comparison evaluating to true doesn't immediately result in a chosen code version, and it is also the only one with horizontal optimisations nested inside another layer of horizontal optimisations. In terms of the simple case tree structures, this represents the most complicated situation.

As for hardware, three hardware configurations are going to be experimented on. These being an Nvidia GTX 780 Ti, an Nvidia RTX 2080 Ti, and finally an Nvidia Tesla K40c. The goal of the tuners is to be able to accurately tune datasets to hardware, thus different hardware types should be used for testing different hardware characteristics. Each GPU represents a different segment of the GPU market, with the GTX representing an older consumer GPU, the RTX representing state-of-the-art in consumer GPUs as of the time of writing, while the K40c represents a compute-focused GPU of the same hardware generation as the GTX card. Comparing the GTX and RTX shows a considerable increase in performance in the 6 year gap of their release dates, with the RTX boasting double the core frequency, cuda cores, and almost four times the VRAM compared to the GTX. The Tesla K40c is not as comparable to either of the two earlier ones, as its primary use is for GPGPU centered data servers, being meant for industrial rather than consumer use [31].

4.1.2 Training Process

No matter the benchmark, in order to tune the rodinia and FinPar programs, a dataset suite have to be supplied for each individually. These datasets are like the benchmark programs also split in two groups, with one dedicated to training, and the other for validation. Validation datasets are unused during training of the tuners, representing future unknown datasets that might arise in real world applications. Doing this helps measure whether the autotuning process is able to tune the thresholds such that they also give good performance on unknown datasets.

The goal of the training datasets is to be representative of the degrees of parallelism that can best utilise the different code versions, and of the datasets that will be used during deployment of the program. In order to fabricate these training datasets, program-specific knowledge has to be applied, but in some cases artificial datasets can work fairly well based on the type signature of the input. Luckily Futhark's type system makes it easy to find the types of the input to any given function, and Futhark also has a handy tool for generating datasets, according to their type along with bounds on the values themselves.

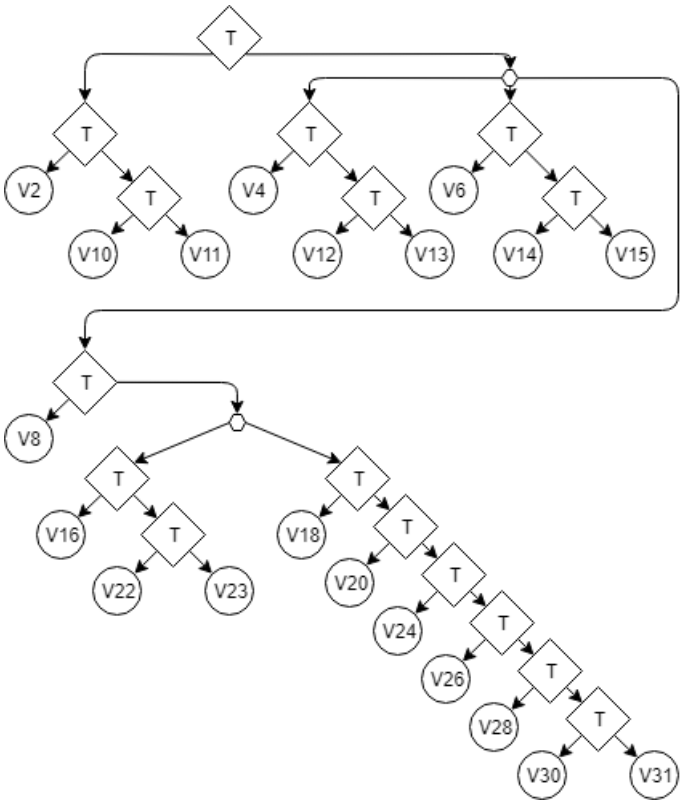


Figure 4.1: *Tree structure of OptionPricing.fut using the legend of Figure 1.8*

To get the best training results, having multiple datasets utilising different degrees of hardware parallelism is ideal. The more varied the datasets, the more nuanced the information that a tuner can extract becomes, and more precise thresholds can be found. This is exemplified by the simple case tuner, whose resulting threshold configuration is entirely derived from values that it has obtained from empirical experiments using the training datasets, meaning it does not try to extrapolate to unknown datasets at all. For example, if a program has 2 training datasets, which both prefer the same code version in the program, then the simple tuner will only ever be able to conclude the degree of parallelism for which that version is good, but nothing about other code versions in the tree. If a new dataset is introduced which is similar to the earlier two, but with a slightly larger $E_{par}(Input)$ value, then the tuner will have no basis for deciding which code version best fits that $E_{par}(Input)$, as it has no experience with that high a value.

Thus, a few training datasets have been supplied for each benchmark to better guide the training process following the above concepts. Each benchmark will have at least two datasets, one being a small dataset, and another being a large one, ideally representing a minimum and maximum degree of parallelism expected. For example, consider SRAD whose `main` function's type signature is as follows:

```
let main [num_images][rows][cols]
         (images: [num_images][rows][cols]u8) :
         [num_images][rows][cols]f32 =
```

This is a `main` function which takes in a three-dimensional array named images of unsigned 8-bit integers, and produces a three-dimensional array of 32-bit floats with the same size. In this situation, a few datasets can easily be created following the above criterias. A small dataset might consist of one image of 16x16 pixels, while a large dataset might consist of one image of large resolution such as 1024x1024. Additional datasets could then be added to also

get information about the importance of the number of images processed. The following calls to `futhark-dataset` show how such artificial datasets can easily be generated in Futhark:

```
futhark dataset --generate=[1][16][16]u8     > srad-D1.in
futhark dataset --generate=[1][1024][1024]u8 > srad-D2.in
```

While creating artificial datasets is a good way of getting a set of input, it can be useful to have a guarantee that the output produced by a GPU program on this new input is correct. This has been accounted for in the training process, by recording the output of running the sequential program for each dummy training dataset, and storing that output for comparison. Futhark contains not only an OpenCL compiler, but also one for sequential execution, which is less prone to error and imprecision as the OpenCL generated code. Commands for doing this is shown below for our SRAD example.

```
futhark c srad.fut
./srad < srad-D1.in > srad-D1.out
./srad < srad-D2.in > srad-D2.out
```

Using the futhark benchmark tool, it can be requested that the output of running a program be compared against one of the output files generated, in order to verify the correct execution of the program. This is done by adding the following lines to the top of the futhark source code, specifying test datasets and their expected output:

```
1 -- ==
2 -- tune compiled input @ srad-D1.in
3 -- output @ srad-D1.out
4 --
5 -- tune compiled input @ srad-D2.in
6 -- output @ srad-D2.out
```

The word "`tune`" is a simple tag used to signify a training dataset, while "`compiled input`" tells the futhark benchmarking tool that the data is in a binary format saved in an external file, which should be used as input for a benchmarking run. The third line then tells the benchmarking tool that whatever result is computed in the second line should be compared against the file `srad-D1.out`. If any value does not match, this will be reported and considered an unusable threshold configuration.

Incremental flattening's code-versions are supposed to be semantically equivalent, but bugs can occur, so having a guarantee of proper execution is good for the artificial datasets. Sometimes this verification process can not be used, since some programs vary in their floating point precision. As an example, some programs might consider a result to be ± 0.5 the sequential value and still be considered correct. If that is the case, a correct result is flagged as incorrect due to imprecision, in which case this check should not have been made. Some benchmarks did have this issue, with their incrementally flattened code versions producing different results than their sequential counter parts, and as such is skipped for some of the training datasets, but is kept for all validation datasets. After all, the only purpose of the training datasets is to provide sample data to run the programs, with their outputs being irrelevant, whereas the validation results have to be correct.

After tuning on the training datasets following the above protocols, the found thresholds are used on the unknown validation datasets. Validation datasets are denoted by the tag "`notune`" in the Futhark file. Their performance increase is the estimator of real world efficiency of the tuners, and in turn of the usefulness of incremental flattening's *"one size does **not** fit all"* optimisation approach.

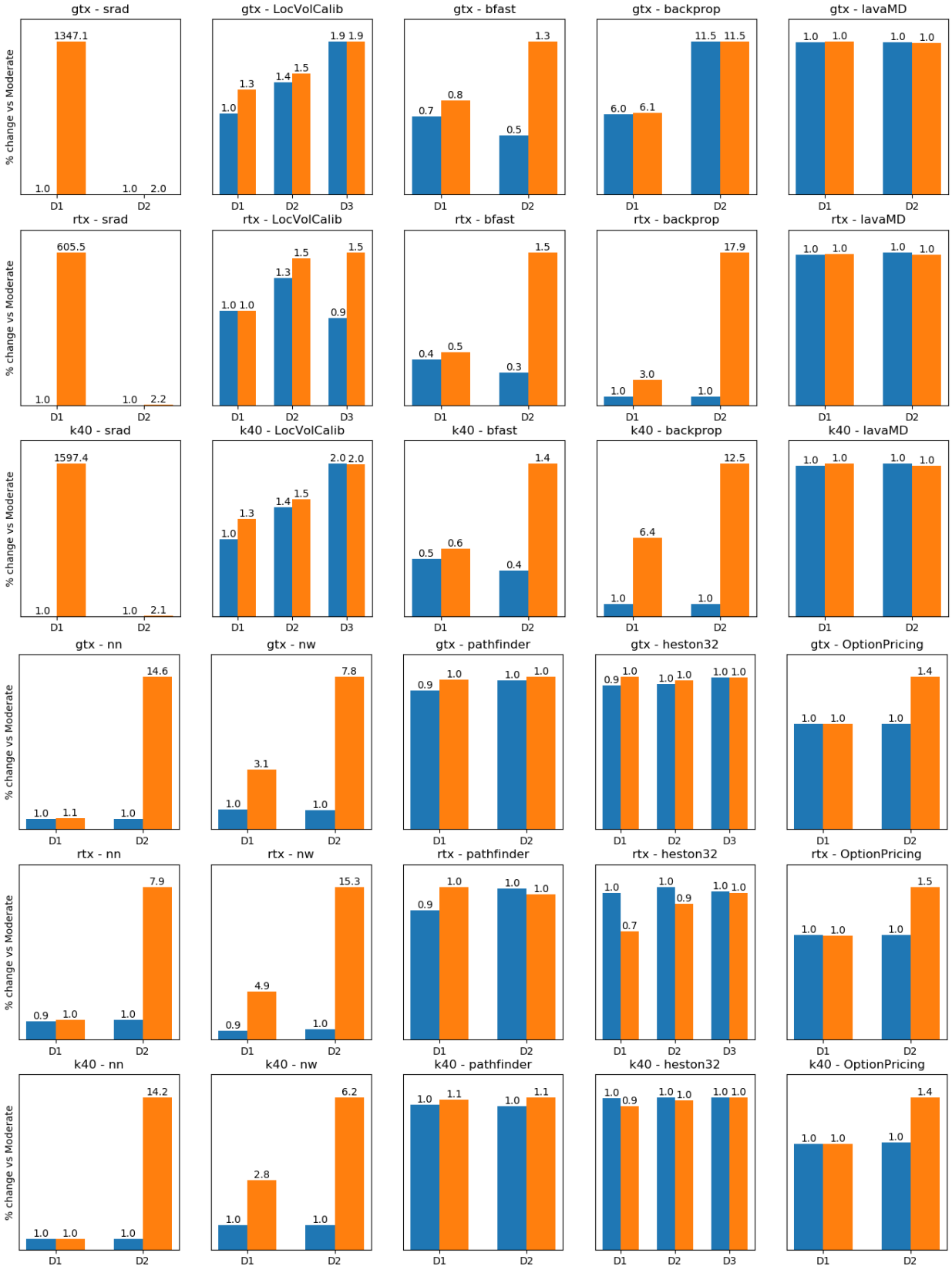


Figure 4.2: *Result of the untuned (Blue) and autotuned (Orange) Incremental Flattening in percentage comparison with moderate flattening (higher is better). The first and fourth row has the GTX results, the second and fifth row has the RTX results and the third and sixth rows are Tesla results. They are split in this order to allow easier comparison of benchmarks across hardware platforms.*

4.2 Results of the Simple Exhaustive Tuner

The simple tuner implementing the pseudocode of Algorithm 2 has been run on all the benchmarks and hardware configurations described in the sections above, with results presented in Figure 4.2. The figure shows the results of untuned and tuned incremental flattening on each dataset as percentage speedup over moderate flattening in Futhark on all three hardware platforms. For each dataset, the benchmark has been run 10 times, and the average runtime reported.

One thing to note is that all of these benchmarks require program specific knowledge to create additional datasets. While it is possible to craft more dummy datasets, in all cases except for `LocVolCalib` this was not done as the datasets provided in the futhark implementations sufficed for both training and validation, though with a few outliers in the form of `bfast`, `heston` and `optionpricing`. For `LocVolCalib`, all training sets were manufactured, as input for this benchmark consists of tuning a few different integer parameters with a few restrictions, making data generation easy.

As for the outliers, `bfast` has 6 training datasets and 2 validation sets which were already provided, with no additional datasets artificially crafted. Of those 8 datasets, the dataset denoted as D1 in Figure 4.2 was the largest of them, potentially running into the same issue as earlier described with the simple tuner not extrapolating to unknown datasets. The `heston32` benchmark likewise had two training sets supplied with three validation sets, where both D2 and D3 are larger than both available training datasets. This particular benchmark's input is structured in a specific way, making artificial dataset generation difficult. Finally, `OptionPricing` falls under the same scenario as `heston32` in that artificial datasets are difficult to manufacture, but where the supplied training and validation datasets are of roughly the same sizes.

A proper discussion of what the results mean in the context of incremental flattening will be had after also presenting the loop based tuner results. The main difference between the simple case results and the loop based ones is that only two benchmarks exist for the latter, with one being an artificially created program. For full reports of runtimes, consult Table 1.

4.3 Results of the Loop-Based Tuners

Most of the above training strategy is the same for the loop-based tuners, with the exception of the benchmark programs used. Standard benchmarking programs using nested parallelism inside of sequential loops are more rare than the earlier cases, and as such the only major program used for validating this tuner is Lower-Upper Decomposition (LUD) also from Rodinia. LUD is an algorithm used to factorise a square matrix into two matrices consisting of the factorisation of the original one, following equation 4.1 and 4.2 :

$$\mathbf{A} = \mathbf{LU} \tag{4.1}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{4.2}$$

Where $\mathbf{A}$ is the matrix we want to factorise, and $\mathbf{L}$ and $\mathbf{U}$ are triangular matrices, which represent the factorisation of $\mathbf{A}$. The algorithm for computing this decomposition contains a

sequential loop over the size of the square matrix, with each subsequent iteration working on a smaller sized sub-matrix of **A**. From this, a parallel implementation involves some form of nested parallelism whose size is tied to a sequential loop, resulting in the difficult structure of trees. Specifically, the structure can be seen in Figure 4.3.

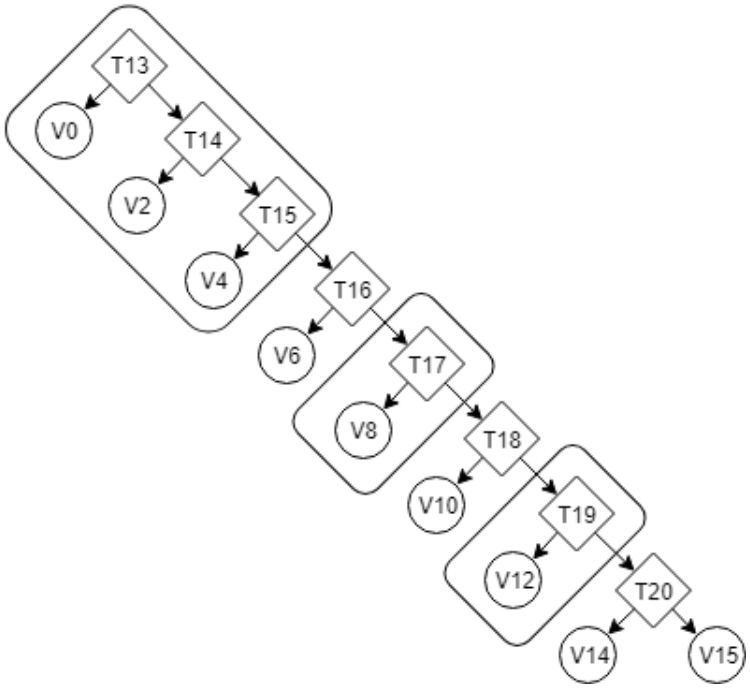


Figure 4.3: *Tree structure of the LUD benchmark. The squares encompassing some parts correspond to those relying on a parallel size changed by a sequential loop, as described in Figure 1.8*

The structure itself is very simple, consisting of a chain with no horizontal optimisations, but with some regular comparisons not dependent on a variant parallel size interspersed. Due to the recursive structure of the tuners described earlier, the simple cases will be tuned exhaustively according to Algorithm 1, while the remaining 5 thresholds will be tuned by the loop-based strategies. For each of the strategies benchmarked, the number of executions run and the total time taken for tuning will be reported. Since no objectively optimal algorithm has been developed, these characteristics will be used to judge which style of strategy performs the best.

As for the datasets used for LUD, nine separate datasets are used, with most being artificially generated. Each one is a square matrix of 32-bit floats, but with varying sizes. Their sizes ranges from 16 by 16 as the smallest, to 4096 by 4096 as the largest. Listing 4.1 shows how these are specified in the futhark LUD file as tests to be benchmarked.
This follows the overall structure of having training datasets representative of the validation datasets, since for every validation datasets two training datasets have slightly higher and lower values of $E_{par}(Input)$. Each one has had it's output verified by the sequential version, as described for the simple cases, except for `256.in`, `1024.in` and `4096.in` which had problems with numerical imprecision for which moderate flattening's baseline did not verify correctly either.

As before, the results are presented in the box plot of Figure 4.4, along with specific runtimes in table 2. The box plot includes specific runtimes from three separate tuning runs

across all three types of hardware. The reason for having a box plot as opposed to a bar plot is to highlight variances in results across independent tuning runs. As opposed to the earlier bar plot for the simple case, the goal here is to first compare the performance of the five algorithms, and then afterwards compare against moderate flattening and untuned incremental flattening in a separate figure. Tuning times for each is also reported in Figure 4.5.

```
-- ==
-- tune input @ data/LUD-data/16.in
-- output @ data/LUD-data/16.out
-- notune compiled input @ data/LUD-data/32.in
-- output @ data/LUD-data/32.out
-- tune compiled input @ data/LUD-data/64.in
-- output @ data/LUD-data/64.out
-- notune compiled input @ data/LUD-data/128.in
-- tune compiled input @ data/LUD-data/256.in
-- output @ data/LUD-data/256.out
-- notune compiled input @ data/LUD-data/512.in
-- output @ data/LUD-data/512.out
-- tune compiled input @ data/LUD-data/1024.in
-- notune compiled input @ data/LUD-data/2048.in
-- output @ data/LUD-data/2048.out
-- tune compiled input @ data/LUD-data/4096.in
```

Listing 4.1: In-file test specification for LUD

Comparisons of the different algorithms will be kept for the following chapter, as the results doesn't quite follow the expected results. In order to gauge the overall quality, the same benchmark has to be run with moderate and untuned incremental flattening. Figure 4.6 shows this, as a bar plot of percentage increases compared with moderate flattening, using the median runtime of each algorithm in Figure 4.4. For specific median runtimes, average tuning time and average number tuning benchmarks run consult Table 6.2 in Appendix B.

The following chapter will discuss what the results from Figures 4.4, 4.5 and 4.6 tell us about the efficiency of the tuners, and whether they live up to the expectations that was set out in Chapter 3.

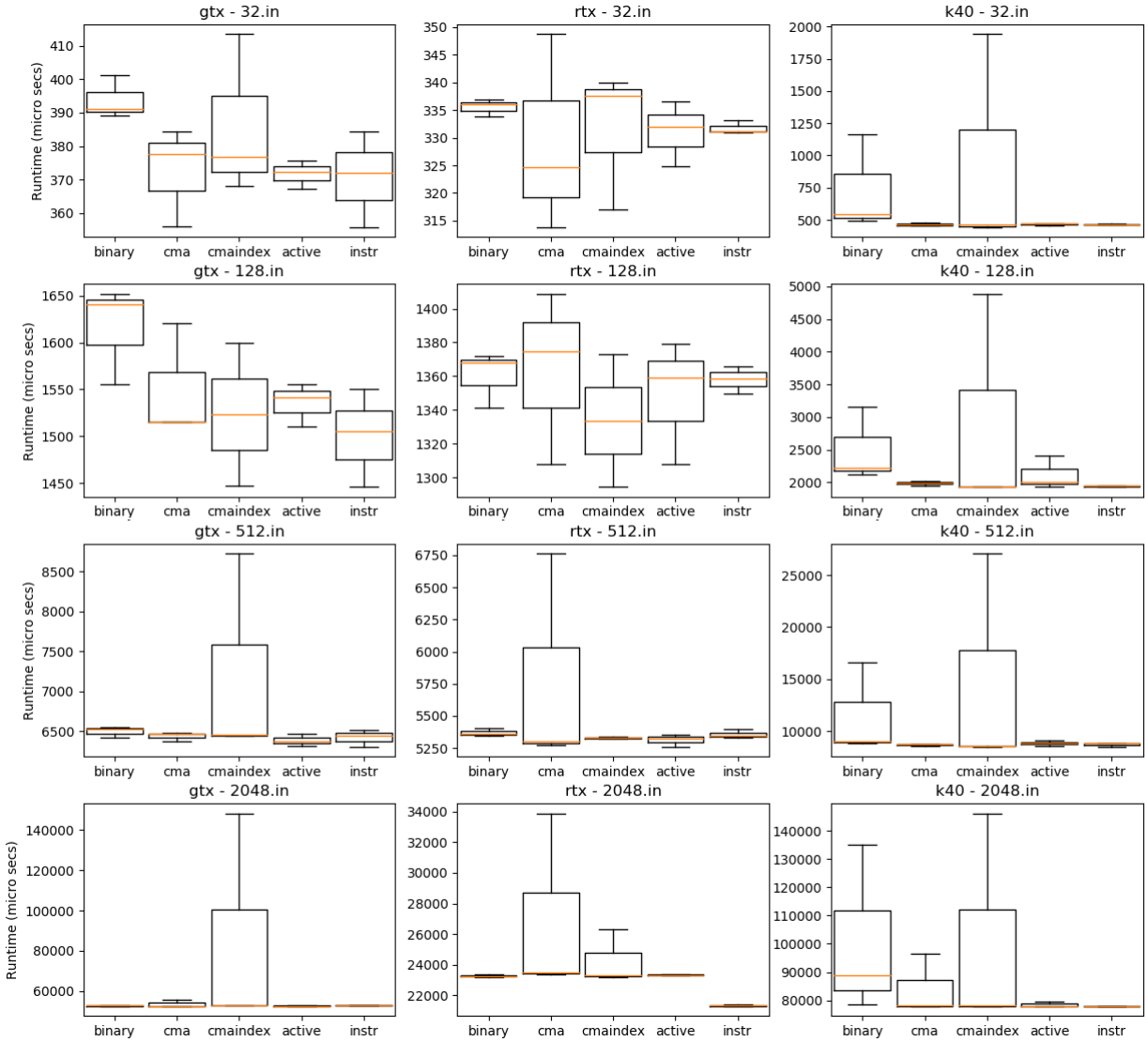


Figure 4.4: *Box plot showing the runtime results of three separate incremental flattening tuning runs, in order to compare all five algorithms.* ***Binary*** *refers to algorithm 5,* ***Active*** *refers to algorithm 6,* `CMA` *refers to the naïve CMA-ES implementation with* `CMAINDEX` *being the changed one where it searches for indexes instead of threshold values. Finally,* `instr` *refers to the instrumentation implementation. The orange line is the median.*

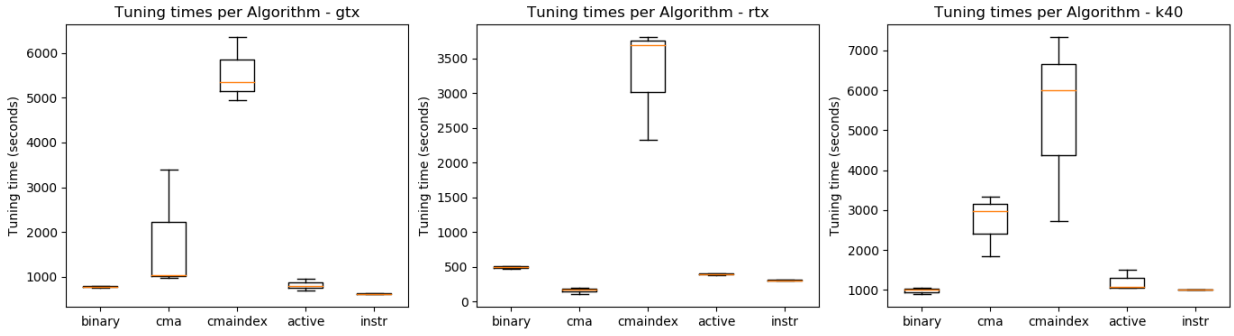


Figure 4.5: *Tuning times for results shown in Figure 4.4 as a boxplot of the three separate tuning runs.*

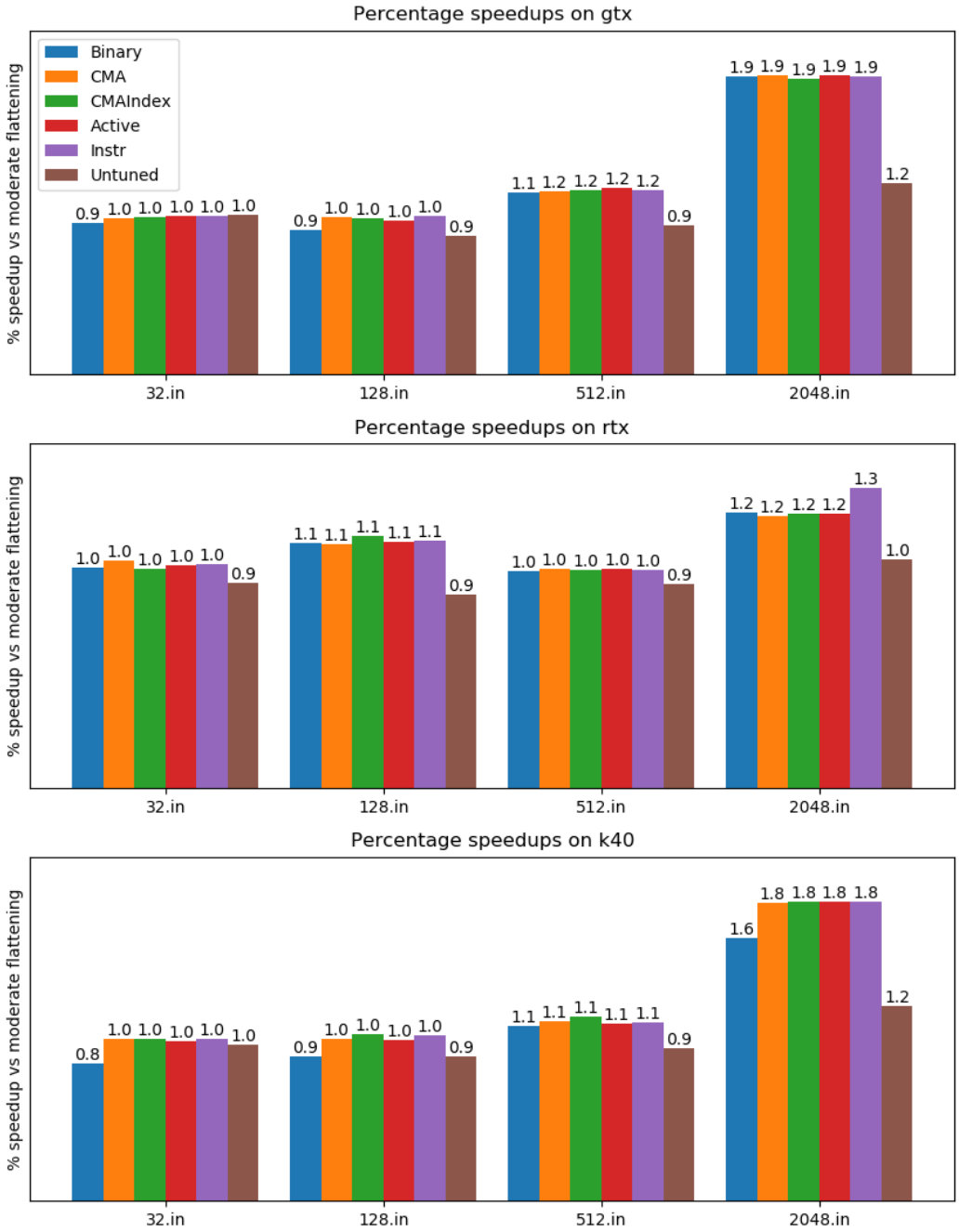


Figure 4.6: *Percentage speedup of each of the five algorithms from Figure 4.4, along with untuned incremental flattening. The percentage is in comparison to Futhark's moderate flattening.*

CHAPTER 5

DISCUSSION

This thesis has covered a few different solution strategies to the autotuning problem, and in this section we will cover each once again on the basis of their actual performance as seen in Figures 4.2 and 4.4.

5.1 Simple case discussion

Algorithm 2 was the first algorithm discussed in this thesis, and serves as the baseline for the remaining ones via Algorithm 3. In terms of the results seen in Figure 4.2, it does seem that given proper training datasets reflecting different degrees of parallelism, tuned incremental flattening provides substantial performance increases, as seen particularly in all benchmarks except `bfast` and `heston32`. The average percentage speedup compared against Moderate Flattening across all datasets (excluding SRAD's massive outlier) and across all hardware is a 3.07x speedup.

In terms of the guarantees discussed when Algorithm 2, they do seem to translate into the experiments from Chapter 4. Most of the problems had some instances where the threshold ranges' intersection were the empty set, but it seemed even in those cases that most of these conflicts could be solved by efficient use of the cached experiments, almost adhering to the guarantee of running only one benchmark pr. code version.

The major drawback that these experiments have uncovered in regards to Algorithm 2 is that relying only on user supplied training datasets leads to problems. Looking at `heston32`, the algorithm's input datasets only differ in the number of quotes that it uses to calibrate it's model, which for the five datasets given is shown in Table 5.1.

	NumQuotes	MaxGlobal	NumPoints	np	today
1062_quotes (D1)	1062	2000	20	40	2017-01-26
train-1162_quotes	1162	2000	20	40	2017-01-26
train-9000_quotes	9000	2000	20	40	2017-01-26
10000_quotes (D2)	10000	2000	20	40	2017-01-26
100000_quotes (D3)	108979	2000	20	40	2017-01-26

Table 5.1: Input sizes for the five datasets supplied for the `heston32` benchmark.

It is clear that they are created such that only the number of quotes to do work on is changed, and as such we can see a few issues with only having the training sets have 1162 and 9000 quotes respectively. Looking back at the results for `heston32`, it is oddly enough D1 and less so D2 being off though they at first glance have most in common with the training datasets, with D3 being on par with moderate flattening. Explaining why D3 is doing well is possible, as all values of $E_{par}(Input)$ for D3 could be high enough such as to completely ignore the thresholds set during tuning, meaning it defaults back to the untuned result which happens to be the same optimisation as moderate flattening. D1 and D2 are more complicated to explain.

One option is for bad results to simply be benchmark noise, and this is possible since it seems to be isolated to the RTX hardware. In order to examine this problem more closely, we uncovered the specific values of $E_{par}(Input)$ used by the different datasets, and it seems that all five datasets has the same values. As an example, both dataset D1 and D3 has all the thresholds compared against the same two values, being either 5 or 40. Looking at table 5.1 seems to indicate that intuitively they should prefer different code versions, and should have different values of $E_{par}(Input)$, but looking at the actual values generated by incremental flattening seems to go against this.

The two repeating values used in threshold comparisons does not reflect the sizes of the input, and we are curious as to how the incremental flattening code generation has reached this situation. Knowing this, what happens in Algorithm 2 to produce the results seen in Figure 4.2 seems to be that the longest running training dataset chooses whichever code version it prefers, and upon having an empty intersecting threshold range that longest running dataset wins out. Since all datasets will always choose the same path due to having the same values of $E_{par}(Input)$, no real training is actually done as all past and future datasets will simply fall into this one code version, which for varied datasets is not ideal.

While it is not uncommon for multiple datasets to have the same values, it is a problem when it happens across all thresholds. As an example of normal behaviour, `backprop` has three thresholds and four datasets. For two of the three thresholds, all datasets use the same value, but for the final one every dataset has a different value, being $[16385, 20481, 839681, 1048577]$ respectively. This allows for accurate clustering of datasets to code versions, and we see very good performance increases in tuned `backprop`.

Finally, the second observation that is of interest in Figure 4.2 is the effect of the different hardware platforms. As described in the methodology, the three GPUs represent vastly different characteristics, and the problem statement of incremental flattening was in part that different hardware might better support varying dataset characteristics [10]. This is in part present in the figure, though not as plainly as was expected.

The benchmark showing this most clearly is `backprop`, in which the untuned and tuned incremental flattening behave much differently between hardware configurations. In general it does seem like the same code versions are preferred across hardware, being the clear case for most, with the only exception being `backprop`. All other benchmarks' percentage increases are roughly the same, with the RTX card consistently having half the percentage increase compared to GTX and the K40c. Considering that the RTX has roughly double the core count and frequencies, it might be explained as the RTX simply being a faster GPU.

Overall, the results for the simple case tuner based on Algorithm 2 are very promising, with the one considerable outlier being explained as a shortcoming of incremental flattening itself. In normal cases, tuned incremental flattening tends to perform at least as well as moderate flattening, but with the possibility of exploiting separate optimisations for different datasets to much better utilise hardware characteristics. While these hardware characteristics do not seem to be as prominent as we expected, the `backprop` benchmark does show that different hardware can prefer different optimisations.

5.2 Loop case discussion

Whereas Algorithm 2 solved the simple case alone, the loop based case is solved by five separate algorithms, each with their own qualities and drawbacks. This discussion is based both on the theoretical expectations of each algorithm based on their presentation in Chapter 3, and how well they compare to the experimental results presented in Figure 4.4, 4.5 and 4.6.

Theoretical expectations of algorithms

The primary focus will be on producing good results consistently, followed by the secondary goal of having a likewise consistently fast tuning time. To reiterate the expectations of each algorithm from before the experimental results, the following will recap each of them.

1. Binary search was expected to have a very reliable tuning time, always requiring $\log_2(N)$ executions as per its guarantee . The results expected from this tuner should also be a good consistent baseline for what can be achieved, as it seemed the assumption of a second degree polynomial held for the validation dataset of LUD. This particular tuner could potentially fall into and get stuck in local minima, but if it does so, it should at least consistently do so across multiple benchmark runs.

2. Evolution strategies, with CMA and CMAIndex being the naïve and more domain-specific implementations respectively. CMA was expected to perform very inconsistent from run to run, with very little options for getting off of individual plateaus in the step function. The tuning time however was expected to be faster in general, due to getting stuck fast and therefore terminating.

   CMAIndex was the extension of CMA to find indexes into the list of possible unique plateaus, in order to give it more meaningful benchmark attempts. As a consequence, the results were expected to be better on average than CMA, at the cost of longer tuning time. Due to being random in nature, it would however not be a consistent tuning time, as it still has the option of randomly finding the optimum in a few random attempts.

3. Active learning was another randomised optimisation strategy, in which the random guesses were chosen based on the notion of an information metric. During development and experimentation, this tuner seemed to be fast at identifying good thresholds, though with the possibility of being quite inaccurate at times due to noise in the benchmarks. As such, while this is expected to have a consistently fast tuning time, the results are not expected to be as consistent due to noise and bad initial sampling.

4. Instrumentation was in Chapter 3 described as the one tuner providing nice guarantees of quality, much like Algorithm 2 does for the simple case. The first of these guarantees was that it required one benchmark execution per code version to get complete information, meaning it should have a very consistent tuning time. Secondly, the results were also expected to be very accurate, with the caveat that the current implementation does not represent the ideal version of this strategy. By this we refer to the inserted GPU barriers and other problems involved in getting the accurate timings, making it extra susceptible to noise.

Expectations compared to experiments

Looking at the results shown in Figure 4.4 seem to indicate that some of the above expectations held, while others did not. Starting at the smallest validation dataset, `32.in`, results across the board were very similar, as the small dataset did not seem to allow for large improvements. This did not stop the CMAIndex and binary search algorithm on the k40 from finding two terrible threshold configurations regardless.

The medium dataset `128.in` on the other hand had very different results, across all three hardware configurations. Binary search was surprisingly inconsistent, being the slowest on the GTX, and having results varying by quite a bit on the GTX and K40 platforms. CMA did still seem the most inconsistent, with active learning also being surprisingly consistent. Finally, the instrumentation tuner performed the most consistently of all.

The second largest dataset, `512.in`, has figures dominated by the three outliers reported by the two CMA-ES, underlying how inconsistent they can be. For the remaining algorithms for this particular benchmark, all three of them are very consistently good. Overall this particular datasets follows the expectations very well, except for binary search's one outlier on the k40.

Moving on to the final large dataset in `2048.in`, few things are unchanged. Binary and active learning are both very consistent, much like before, except for two outliers on the K40c. CMA and CMAIndex are again highly inconsistent in their results, which has been the case across all datasets. Finally, the instrumentation tuner provides the best results in this dataset, while also being the most consistent. Particularly on the RTX platform does this show, as the instrumentation tuner finds the best code version in all three tuning runs, a code version which is not found by any other algorithm. It should be noted that the `2048` dataset has output verification, meaning it a valid threshold assignment.

With the primary goal of actual results discussed, Figure 4.5 show the tuning time required for each of the runs for the three platforms. While the runtime results themselves did not always follow expectations, tuning time is more or less as expected. The three algorithms which are less stochastic, being binary search, active learning and the instrumentation tuner all follow the same pattern consistently. Binary search and active learning are roughly equal, with both being beaten slightly by instrumentation. The CMA-ES based tuners both vary more in their tuning times, with very different tuning times across the three platforms. Both do follow expectations somewhat, with the more involved indexed version getting less stuck, leading to longer tuning. This is especially true on the K40c, in which it reaches its' timeout value of 2 hours tuned.

Comparing these results with those of moderate flattening and not doing any tuning of incremental flattening gives a bit of context to the results. For all datasets except the smallest one in Figure 4.6, performance increases were found across all hardware. In particular, the largest dataset had the most noticeable increase in performance, which makes sense as it has the most chance of seeing benefits from switching between code versions between loop iterations. Once again the best tuner on average was the instrumentation tuner, which compared with moderate flattening experienced an average speedup of 1.2x.

Takeaways from the loop based case

Choosing a single tuner as the one to "solve" the tuning problem is difficult, as different trade offs exist. If consistency is important, the active learning and instrumentation tuner both provide very consistent tuning times, with near-optimal results. However, due to the theoretical guarantees and the random nature of active learning, the instrumentation tuner still seems to be the best choice both in theory and experiments.

In addition, instrumentation even allows for more enhancements, and the implementation used for these experiments does leave room for improvement. Particularly the proper implementation of timing code for specific code versions individually, rather than for every kernel as is done here, would ideally help make it less prone to errors in it's results. Likewise, implementing early stopping to stop running useless benchmarks once the best result is found could also improve performance in terms of tuning time even further.

With both of these improvements left for future work, the results for the loop based case still speaks in favour of the instrumentation tuner being the best all-round solution to the tuning problem. That is not to say that the other tuners are not of interest, as most of them also has plenty of opportunities for enhancements, though from the experiments we doubt they will surpass the instrumentation tuner.

One major issue with the experiments for this loop based case is that of having only one benchmark program in LUD from rodinia. It is a more rare case, but having a single validation program is not sufficient to draw any conclusive results. Experimentation was done also on the manufactured dummy example from Listing 1.6 from Chapter 1, but as it was used extensively during development of each algorithm, including results from said example would run the risk of having each algorithm overfitted to that specific problem. For instance, the active learning strategy chosen was decided entirely through experimentation on that program, so it should not be included in validation of the tuners. For future work, it would be ideal to introduce additional test examples to support the results from Figures 4.4, 4.5, and 4.6.

5.3 Future Work

While this thesis provides a solid foundation for tuning Futhark's incremental flattening programs, it does still have multiple areas left for future work. These range from improving specific parts of the tuners, but also about additional parameters to be tuned such as tile sizes, and general changes that could be made. Here we will present a list of such areas that is left for future work, that could improve upon this thesis.

Improving the instrumentation tuner

Improvements on the instrumentation tuner seems the best avenue for further progress in producing a stable tuner with good guarantees. The two major improvements would consist of the following:

- Proper instrumentation to insert timing code around code versions.
- Early stopping when the optimum is detected.

With both of these, the results will be more stable, while also improving tuning time even further. As the results from the loop based experiments showed, the tuning time of the flawed instrumentation tuner was already the best, but in theory there is definitely still room for improvement.

Detection of register and block tiling

Tuning for additional parameters such as tile or workgroup sizes is currently lacking, even though it is a considerable part of tuning for specific code versions. Under all the tuners implemented, the tuning of tile size has been done in a single pass over a few values, meaning nothing substantial has really been tuned. The reason for this is, as mentioned earlier, that it would explode the search space if done improperly, while only a few specific programs with code versions using block or register tiling would actually benefit from proper tuning.

Detecting whether a code-version uses additional parameters outside of thresholds would allow much more precise tuning. For example, every time a different tile size would be attempted under the old implementation, every code version would have to be retried using this new parameter, even if said code version doesn't use it. If we knew specifically which code versions would benefit from having said parameters, only relevant code version benchmarks would be run.

Shrinking the search space like this is the first step towards being able to tune tile sizes. Looking at the instrumentation tuner, it was shown that shrinking the search space was a major part of reducing tuning time from the previous solutions. Implementing this has not been done, as it is outside the scope of this project to modify the Futhark compiler.

New active learning strategy

While the active learning tuner was outperformed by the instrumentation tuner, it was definitely capable of solving the problem in an efficient way. The active learning tuner as was implemented for the experiments could definitely be expanded upon, which is also an area of future work.

One such idea could be to implement the use of gradient descent, along with its many modifications available, to guide the search for good unlabeled datapoints. The simplest way of doing this could be to apply gradient descent to the predicted second degree polynomial model, to guide for predictions. Likewise, gradient descent could also be used as a replacement for the binary search strategy.

This was not attempted, as the active tuner already was capable, but we believe it is possible to further expand it, in ways such as just described.

Step Function optimisation

The active learning and CMA-ES functions both work primarily on continuous functions, even though the actual objective function is a step function. This has proved problematic, especially for CMA-ES which has trouble finding proper correlation between changing threshold values and where the plateaus change. Both use tricks to convert the solution space into a smaller one, but not always successfully.

It would be interesting to try optimisation methods focused on discrete optimisation instead of continuous, such as particle swarm optimisation. Swarm optimisation most closely resembles an evolution strategy, but with one population of individuals each exploring the discrete space separately, without selection and replacement of individuals in comparison to CMA.

Besides trying an entirely new optimisation strategy, both active learning and CMA-ES could benefit from changing their prediction model and fitness functions respectively to better reflect the actual objective function. While this part of the future work doesn't have a clear value in terms of better performance, it is an area that is lacking in this thesis which could have potential.

Rewrite of tuners to work in Haskell

Futhark's tools are all written using Haskell, and the logical extension to employ the work in this thesis would be to convert it to Haskell. The implementation whose experimental results were presented in Chapter 4 are built in Python, and as such is not an ideal fit with Futhark's backend tools.

5.4 Conclusion

Incremental flattening provides flexible choices for individual datasets to pick from multiple semantically equivalent optimisations. This thesis has shown both the considerable benefit of exploiting the flexibility, and that threshold tuning is doable, even with few training datasets to learn from.

Tuning thresholds to correctly cluster datasets to optimisations can be done in a multitude of ways, each with different trade-offs in terms of tuning time and result quality. While some tuners did not perform well in general in terms of solving the tuning problem, the instrumentation tuner in particular provided solid theoretical properties that translated into experimental results. Together with the tuner for the simple case, most benchmark programs yielded solid improvements, averaging 3.07x speedup in the simple case and 1.2x in the difficult case for LUD.

Combining incremental flattening with a consistent tuner was shown to vastly improve performance in commonly used benchmark programs. This allows for the Futhark programming language to generate even faster code for programmers not familiar with GPU programming, further bridging the difficulty gap that has so far hindered GPGPU programming.

CHAPTER 6

APPENDIX

Appendix A - Learning Goals & Tasks

Learning Goals

The learning goals as they were expected from the onset of this thesis:

- Understand the concepts of auto-tuning in relation to parallel programming, and how it is implemented in OpenTuner used by the current Futhark compiler.
- Improve upon the current Futhark auto-tuning approach, in order to further utilise the parallelism in the different code-versions produced by Futhark. This could potentially include proper integration of OpenTuner's OpenCL parameters, which is not currently done.
- Experiment with the benefit of applying concepts from Machine Learning to aid the off-line learning tasks of estimating good threshold values.
- Measure the impact of the different experiments in terms of different cost-functions for the auto-tuner benchmarks.

Tasks

In order to achieve the learning goals, I was expected to follow the following structure:

- Analyze and get familiar with the current auto-tuner for Futhark's thresholding parameters, and in particular how OpenTuner is used.
- Extend the existing tuner, in particular trying to continue with the current approach of estimating thresholds based on defining a search space and iterating through it.
- Implement an alternative tuner based on suitable Machine Learning for the setting, with the goal of learning the underlying function describing the degree of parallelism associated with the threshold values.
- Design concrete experiments to gauge the advantages and disadvantages of the two strategies, using benchmarks based on code of varying degrees of complexity.

Appendix B - Tables of benchmark results

		SRAD		LocVolCalib			Bfast	
		D1	D2	D1	D2	D3	D1	D2
Nvidia GTX 780 Ti	Moderate	16.5s	29.9ms	130ms	165ms	2.33ms	37.3ms	31.4ms
	Untuned	16.5s	30.1ms	129ms	118ms	1.23s	57.1ms	63.1ms
	Tuned	12.2ms	14.8ms	99.7ms	110ms	1.23s	47.3ms	24.5ms
Nvidia RTX 2080 Ti	Moderate	5.03s	16ms	69ms	43.5ms	590ms	11.7ms	11.2ms
	Untuned	5.03s	15ms	72.7ms	33.9ms	667ms	26.8ms	35.7ms
	Tuned	8.3ms	7.4ms	72.5ms	29.3ms	382ms	23.1ms	7.68ms
Nvidia Tesla K40c	Moderate	26s	43ms	193ms	252ms	3.64s	51.8ms	46.8ms
	Untuned	26s	43.6ms	194ms	179ms	1.84s	95.8ms	109ms
	Tuned	16.3ms	20.2ms	153ms	167ms	1.85s	81.2ms	32.6ms

		Backprop		LavaMD		NN		NW	
		D1	D2	D1	D2	D1	D2	D1	D2
Nvidia GTX 780 Ti	Moderate	1.89ms	148ms	2.48ms	0.37ms	4.54ms	18.3ms	234ms	106ms
	Untuned	0.32ms	12.9ms	2.47ms	0.37ms	4.55ms	18.3ms	232ms	109ms
	Tuned	0.31ms	12.9ms	2.46ms	0.37ms	4.25ms	1.25ms	75.7ms	13.5ms
Nvidia RTX 2080 Ti	Moderate	0.45ms	28ms	0.51ms	0.19ms	2.96ms	5.28ms	210ms	112ms
	Untuned	0.45ms	28ms	0.51ms	0.18ms	3.16ms	5.25ms	240ms	111ms
	Tuned	0.15ms	1.56ms	0.51ms	0.18ms	2.94ms	0.67ms	43.2ms	7.29ms
Nvidia Tesla K40c	Moderate	2.64ms	221ms	3.36ms	0.48ms	5.92ms	24.9ms	325ms	121ms
	Untuned	2.61ms	221ms	3.42ms	0.49ms	6.06ms	24.8ms	326ms	121ms
	Tuned	0.41ms	17.7ms	3.37ms	0.49ms	5.95ms	1.75ms	114ms	19.4ms

		Pathfinder		Heston			OptionPricing	
		D1	D2	D1	D2	D3	D1	D2
Nvidia GTX 780 Ti	Moderate	0.86ms	0.86ms	28.7ms	72.6ms	593ms	11.4ms	9.04ms
	Untuned	0.92ms	0.86ms	30.4ms	76.1ms	597ms	11.3ms	9.05ms
	Tuned	0.86ms	0.84ms	28.7ms	74.3ms	595ms	11.4ms	6.26ms
Nvidia RTX 2080 Ti	Moderate	0.72ms	0.72ms	19.9ms	29.7ms	138ms	2.47ms	5.38ms
	Untuned	0.84ms	0.71ms	20ms	28.8ms	138ms	2.48ms	5.38ms
	Tuned	0.71ms	0.74ms	27ms	32.2ms	139ms	3.69ms	2.5ms
Nvidia Tesla K40c	Moderate	1.2ms	1.22ms	31.3ms	103ms	917ms	15.5ms	12.4ms
	Untuned	1.18ms	1.2ms	31.5ms	103ms	917ms	15.5ms	12.3ms
	Tuned	1.14ms	1.13ms	33.1ms	105ms	919ms	15.5ms	8.59ms

Table 6.1: Specific runtimes reported in Figure 4.2, measured in microseconds or seconds.

		LUD Dataset Runtimes				Tuning Time (s)	Tuning Benchmarks
		32.in	128.in	512.in	2048.in		
Nvidia GTX 780 Ti	Moderate	0.37	1.49	7.46	98.6	0	0
	Untuned	0.37	1.71	7.97	82.0	0	0
	Binary	0.39	1.64	6.52	52.7	778	45
	CMA	0.38	1.52	6.47	52.6	1807	131
	CMAIndex	0.38	1.52	6.45	53.1	5548	233
	Active	0.37	1.54	6.38	52.6	815	42
	Instr	0.37	1.51	6.45	52.8	621	18
Nvidia RTX 2080 Ti	Moderate	0.33	1.49	5.18	28.4	0	0
	Untuned	0.36	1.74	5.72	28.0	0	0
	Binary	0.34	1.38	5.36	23.3	493	45
	CMA	0.32	1.37	5.3	23.5	158	9
	CMAIndex	0.34	1.33	5.33	23.3	3275	173
	Active	0.33	1.36	5.33	23.3	394	32
	Instr	0.33	1.36	5.35	21.3	308	18
Nvidia Tesla K40c	Moderate	0.46	1.97	9.63	142	0	0
	Untuned	0.48	2.23	10.3	120	0	0
	Binary	0.54	2.22	9.02	88.8	980	45
	CMA	0.46	1.99	8.77	78.1	2715	109
	CMAIndex	0.46	1.93	8.57	78.1	5355	99
	Active	0.46	1.94	8.8	77.9	1208	37
	Instr	0.46	1.97	9.63	143	1011	18

Table 6.2: Specific median runtimes reported in Figures 4.4 and 4.6, measured in milliseconds. The average times from Figure 4.5 to tune each strategy is reported as well, along with average number of benchmarks run during tuning.

Appendix C - User Guide

The implementation of the tuners was done in Python, and can be found in the following GitHub repository:

`https://github.com/WizardWithoutHat/Futhark-Autotuner`

It is structured in the following way:

- The root directory contains all the different tuners in separate Python files.
- The `loop-benchmarks` directory contains the loop-based benchmarks.
- The `simple-benchmarks` directory contains the simple-case benchmarks.

In each of the two latter directories, two makefiles exist to run experiments more easily. In order to run the python code, the following packages are required, alongside the built-ins:

`numpy, sklearn, opentuner, pycma`

It is also a requirement to have Futhark installed, as it will be called in the python programs to run benchmarks. In order to tune a program using a tuner, the futhark file to be tuned needs inline training and validation datasets specified using the tags "`tune`" and "`notune`" respectively. To tune, run the following command:

`FUTHARK_INCREMENTAL_FLATTENING=1 python ALG-tuner.py program1 program2`

Where `ALG` is one of `simple`, `binary`, `cma`, `cma-index`, `active`, or `instr`. The tuners can tune multiple programs in one program call, meaning it will first tune `program1` followed by `program2`. Be advised, if the tuners crash due to a problem with `program1` then `program2` will not be executed.

Alongside this thesis, a zip-file named `tuners.zip` containing the tuners has been submitted. Validation benchmarks and corresponding datasets can be found on the github above, but were too large to submit.

BIBLIOGRAPHY

[1] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. Department of Computer Science, University of Copenhagen, November 2018.

[2] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[3] Mike Fritze, Patrick Cheetham, Jennifer Lato, and Paul Syers. The death of moore's law. `http://www.potomacinstitute.org/steps/images/PDF/Articles/FritzeSTEPS_2016Issue3.pdf`, 2016.

[4] Richard Vuduc and Jee Choi. A brief history and introduction to gpgpu. In *Modern Accelerator Technologies for Geographic Information Science*, pages 9–23. Springer, 2013.

[5] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, pages 3–10, 2010.

[6] Sain-Zee Ueng, Melvin Lathara, Sara S Baghsorkhi, and W Hwu Wen-mei. Cuda-lite: Reducing gpu programming complexity. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. Springer, 2008.

[7] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. *ACM SIGPLAN Notices*, 46(8):47–56, 2011.

[8] Andrew Stromme, Ryan Carlson, and Tia Newhall. Chestnut: A gpu programming language for non-experts. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 156–167. ACM, 2012.

[9] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *ACM SIGPLAN Notices*, volume 52, pages 556–571. ACM, 2017.

[10] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPOPP '19, pages 53 – 67. ACM, February 2019.

[11] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[12] Cosmin Oancea. *Lecture Notes for the Software Track of the Programming Massively Parallel Hardware Course at DIKU*. September 2018.

[13] Guy E Blelloch. *NESL: a nested data parallel language*. School of Computer Science, Carnegie Mellon University Pittsburgh, PA, 1992.

[14] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 49:31, 2012.

[15] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[16] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[17] Guy E Blelloch and Gary W Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of parallel and distributed computing*, 8(2):119–134, 1990.

[18] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and GPGPU performance of futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 17–24, New York, NY, USA, 2016. ACM.

[19] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[20] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *computer*, 27(6):17–26, 1994.

[21] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[22] Nikolaus Hansen. The cma evolution strategy. `http://cma.gforge.inria.fr/`, December 2016.

[23] Christian Igel. Cma-es for reinforcement learning slides in advanced topics in machine learning. October 2018.

[24] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Proceedings of IEEE international conference on evolutionary computation*, pages 312–317. IEEE, 1996.

[25] Christian Igel, Thorsten Suttorp, and Nikolaus Hansen. A computational efficient covariance matrix update and a (1+ 1)-cma for evolution strategies. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 453–460. ACM, 2006.

[26] Nikolaus Hansen, Youhei Akimoto, and Petr Baudis. Cma-es/pycma on github. *Zenodo, DOI*, 10, 2019.

[27] Futhark 0.12.0 documentation: Basic usage. `https://futhark.readthedocs.io/en/latest/usage.html`. Accessed 31. July 2019.

[28] Norm Matloff. Programming on parallel machines. `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`, December 2017. Accessed 29. July 2019.

[29] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[30] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E Oancea. Finpar: A parallel financial benchmark. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):18, 2016.

[31] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.