
Bachelors project

Autotuning Futhark

Simon Rotendahl (mpx651) & Carl Mathias Graae Larsen (pwh334)
{*simon, cala*}@di.ku.dk

Contents

1	Introduction	1
2	Background	1
2.1	Futhark	1
2.2	Nested parallelism	2
2.3	Moderate flattening	2
2.4	Incremental flattening	3
3	Autotuning	5
3.1	Search space and portability	7
A	Code examples	9
A.1	Matrix Multiplication - CUDA for GPUs [9, p. 71]	9

Abstract

1 Introduction

2 Background

A common way to increase computer performance, is to increase the capacity for parallelism. For practical usage, however, this is difficult to implement, due to low-level GPU-specific languages requiring domain specific knowledge to make full use of that capacity. A vast amount of work has gone into transforming high-level hardware-agnostic code into these low-level GPU-specific languages [7].

2.1 Futhark

The programming language **Futhark** aims to solve this problem. The creator of Futhark writes the purpose, of the language, nicely on the home page for the language *"Because it's nicer than writing CUDA or OpenCL by hand!"* [4]. On the same page, Futhark is described, more precisely, as *"a statically typed, data-parallel, and purely functional array language"*, but better than a description, is an example:

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

Listing 1: Matrix-matrix multiplication in Futhark [5]

A Futhark program for matrix-matrix multiplication can be seen in listing 1, the syntax is similar to languages such as ML, and Haskell. It is a good example of how Futhark differs from CUDA or OpenCL (we would have liked to include an example of CUDA, but it was too long, so see A.1 for that). It allows the programmer to write efficient parallel code, without all the domain specific knowledge regarding massively parallel systems.

The reason why Futhark is a functional language, is that functional languages lend themselves well to parallel problems, due to the functional paradigm not relying on evaluation order and side effects, as much as other many other paradigms. An example of this is `map`, which can process each element in parallel. However there are still issues, with the main one being that one parallel thread may not spawn more threads, also called *flat* parallelism. Due to this issue, among others like too small of a stack and no function pointers, a functional language by itself is not a solution, and modifying an existing language such as Haskell is not optimal due to the size and expressiveness of it, being poorly suited for the restrictive nature of a GPU.

Many problems are not flat. Take the `dotprod` function from listing 1. Here there is a `map2`¹ which is nested inside of a `reduce`, both operations are parallel, thereby giving *nested parallelism*. This is the main issue mentioned before, and mapping this nested parallelism into efficient flat parallelism is the most important function performed by the Futhark compiler.

2.2 Nested parallelism

Nested parallelism is when there are parallel constructs nested within each other, a simple example is a perfect loop nest, e.g. nested loops with all the work at the innermost level of the nest.

Futhark supports such nested parallelism. More specifically, Futhark supports nested *data-parallelism*. Data-parallelism is when some function/operation, is applied to a dataset in parallel, such that one thread uses function `x` on a subset of data `y`, and another thread also uses function `x`, but on a different subset of `y`. Other languages/environments that support nested data-parallelism are the likes of; CUDA, Open MP, OpenCL, NESL etc. With NESL [2] being the conceptual precursor to the other languages. NESL solved the issue of nested data-parallelism by flattening data structures (both regular and irregular), by a process called flattening [1]. NESL solution however is often inefficient, due to it always maximizing available parallelism, which can lead to overhead, that is greater than exploiting less of the parallel construct. It uses a lot of memory, due to copies of the data, and finally, locality-of-reference optimizations are basically impossible due to poor access pattern information [3].

2.3 Moderate flattening

In the world of GPU's we have *kernels*, a kernel is simply a GPU program. In CUDA specifically, it allows the programmer to define a function, that, when called, is executed N times in parallel by N different *CUDA threads* or *CUDA blocks*². Such a function is called a kernel, and an example of a CUDA kernel is the matrix-matrix multiplication in appendix A.1.

A kernel can be thought of as the work, that is to be done, in a perfect *parallel nest*. A perfect parallel nest is a nest where all the work inside the nest, is at the innermost level of the nest. An example of nested data-parallelism and transformation to kernels, can be seen in figure 1. We can see that the outer `map`, of listing 2, contains more parallelism in it, in terms of the parallel constructs `reduce` and `map`, so in the case where the outer `map` does not saturate the GPU, we would want to exploit the parallelism of the `map` body. With listing 2 this is not possible, since the outer `map` would be parallelised, and the inner sequentialized to one GPU kernel. Instead we can transform it to listing 3. Here the SOACs of the body (`map` and `reduce`) in listing 2, are distributed out into their own `map` nests, giving two perfect `map` nests, and thereby translates to two GPU kernels,

¹`map2` is a variant of `map` taking two arrays instead of a single element and an array. For example `map2 (+) [1,2,3] [4,5,6] → [5,7,9]`

²For a further explanation see threads and blocks see [9]

```

1 map (\xs -> let y = reduce (+) 0 xs
2     in map (+y) xs)
3   xss
4

```

Listing 2: Code fragment before distribution.

```

1 let ys = map (\xs -> reduce (+) 0 xs)
2   xss
3 in map (\xs y -> map (+y) xs) xss ys
4

```

Listing 3: Code fragment after distribution.

Figure 1: Two version of a fragment of code, showing the distribution of SOACs (*second-order array combinators*) [3].

where the first kernel of line 1, is passed into the second kernel of line 2, allowing for further exploitation of the nested parallelism.

This transformation is dubbed *moderate flattening* [6], due to its conceptual resemblance to the flattening algorithm put forth by Blelloch and Sabot [1]. However it is moderate in its approach, due to it flattening the parallel construct until the parallelism saturates the hardware, after which it efficiently sequentialize the remaining parallelism. While flattening supports both regular and irregular structures, Futhark only supports regular structures, and will sequentialize the irregular structures (however Futhark, and its creator documents how to optimize for regular structures). As we saw in Figure 1, Futhark flattens nested parallel construct by extracting kernels, based on rewrite/flattening rules. At the time of implementing moderate flattening, the algorithm used to apply these rules were based on heuristics about the structure of `map` nest contexts. These were good approximations but due to the nature of GPU's (having different capacities for parallelism) there is no one size fits all.

2.4 Incremental flattening

Moderate flattening relied on heuristics, to appropriately flatten the parallelism in a Futhark program, however, as mentioned, this is a good approximation but it lacks the ability to dynamically flatten appropriately to the hardware capabilities, and the size of the data being worked on. This is where incremental flattening comes in. In short incremental flattening statically generates multiple different, semantically equivalent, piecewise code versions of the same program, based on the rules of moderate flattening (along with some additional ones) [7], and then dynamically chooses whether or not to further exploit parallelism or not.

To get an intuition for incremental flattening, let's go back to the matrix-matrix multiplication example in listing 1. Due to matrix-matrix multiplication containing a lot of `maps` (as do many GPU programs), we will briefly explain the code versions generated by the flattening rule, when a `map` containing additional parallelism (CV is short for code version);

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

thread executing the map body sequentially.

- CV1 The body of the map is partially executed in parallel. This means that there will be assigned one GPU group/block to each element of the array being mapped over. This is partial since it does not exploit the parallelism fully, due to not having enough threads to fully parallelise the entire construct within the `map` body.
- CV2 Continues to flatten the `map` function, since we have still have further parallel capacity to exploit.

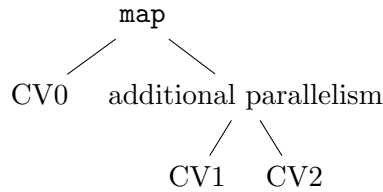


Figure 2: A tree showing the structure of the three different code versions generate by flattening a `map` nest

In matrix-matrix multiplication, there are three levels of nested parallelism to exploit; the outer and inner `map` on line 5, and the `dotprod` function. Lets look at the different ways we can execute the function, there are at least 5 different ways to execute the code, generating 5 different code versions (denoted by a V) [7];

- V0) The outer `map` is distributed out across the parallel construct, executing the body of the function sequentially. More specifically one GPU thread calculates one row in the resulting matrix each. This is CV1 for `map`.
- V1) The outer `map` is executed in parallel, and the inner `map` of `main` is executed partially in parallel. More specifically one GPU group/block calculates one row in the resulting matrix. This is CV2 for `map`.
- V2) The two outer maps of `main`, are extracted as kernels and the kernel of the outer `map` will invoke the inner `map` kernel saturating the GPU, and the body of the two maps (`dotprod`) is executed sequentially. More specifically each GPU thread is calculating one element in the resulting matrix. This is CV1 for `map`.
- V3) The two outer maps, does not saturate the GPU, but the `dotprod` function would oversaturate it, therefore is `dotprod` partially executed in parallel. More specifically each element of the result matrix is executed by a GPU group/block. This is CV2
- V4) If the entire parallel nest does not oversaturate the GPU, the entire nest is executed in parallel³

³Reduce does not appear to be a parallel construct, due to the elements being dependent on each other, however this is done by a *segmented reduce* [8]

We have to choose between these different, but semantically equivalent, ways of execution. The optimal choice will depend on the hardware, and the data being worked on. To visualize the choices, we will look at them as a tree. In Figure 3 there are four choices, we have to consider, and these choices are dependent on each other

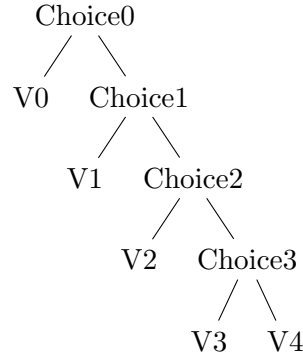


Figure 3: The structure of choices found in the futhark program for matrix-matrix multiplication (see listing 1). (V) represents the resulting code version of a choice

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

3 Autotuning

The process of autotuning a Futhark program is, to automatically pick the fastest combination (or single) code versions, based on the hardware the program is executed on, and some representative data. In the process of selecting the appropriate code version we have three important terms we, that we be central to this report, and therefore we wish to clearly define them in the context of this report;

Threshold or just *threshold* is a value that symbolizes the capacity for parallelism **parameter** (memory, thread count etc.), of the hardware the program is executed on.

Dataset value is a value that represent some of the dataset given. For example for the outer `map` in matrix-matrix multiplication, the dataset value would (primarily) be constructed by the number of rows.

Predicate threshold comparison, or just comparison, is a comparison between a threshold and a dataset value that guard which code version to use.

To pick the best version, we need to tune the threshold parameters, to values that will result in the combination of code versions that gives the fastest runtime. This process is called tuning. The guarding predicates are of the form *threshod* \leq *dataset value*. By default the threshold parameter is set to a value off 2^{15} , as an estimate, but is likely sub-optimal. With these terms, we can now fill out the example in Figure 3, getting Figure 4.

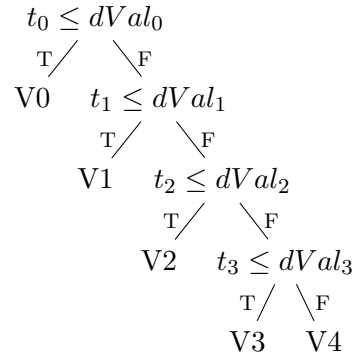


Figure 4: The tree generated by matrix-matrix multiplication that is to be tuned. (t_i) is a threshold, $(dVal_i)$ is a dataset value, and (T) and (F) are indicative of the path we take, based on the threshold comparison.

To inspect the structure of these predicates and thresholds parameters further, lets look at a more complex Futhark program, from Futhark bench, called *LocVolCalib*;

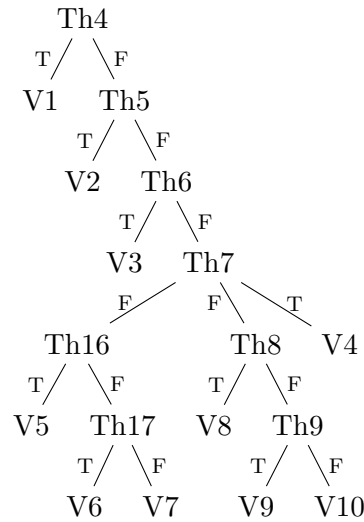


Figure 5: The dependencies between thresholds, of the test program *LocVolCalib.fut*. (Th) is a threshold comparison, (V) is a code version, and (T) and (F) are indicative of the path we take, based on the threshold comparison

To tune a program we need to examine each execution path through the tree. It is

important not to get an end node (code version) confused with a version of the program. Two example of paths through the tree in Figure 5, that shows this, could be;

- $\{(\text{Th4}, \text{False}), (\text{Th5}, \text{False}), (\text{Th6}, \text{False}), (\text{Th7}, \text{True})\}$
- $\{(\text{Th4}, \text{False}), (\text{Th5}, \text{False}), (\text{Th6}, \text{False}), (\text{Th7}, \text{False}), (\text{Th8}, \text{False}), (\text{Th9}, \text{True}), (\text{Th16}, \text{False}), (\text{Th17}, \text{True})\}$

The first path is simple, the code represented by T4, T5, T6 is executed in parallel, where everything after it, is executed sequentially. The second path is more interesting, T7 has two child nodes, that are reached with a false comparison. Here it is clear that two end nodes are reached, namely (V6, V9), and these two code versions are then combined into one program. This is also important to note, because we could have a forest, instead of a single tree, and this would leave multiple code versions, that are to be combined.

3.1 Search space and portability

There are two main reasons for the use of autotuning (compared to manual tuning), the search space of different threshold settings, and the portability of the code.

The search space for thresholds is quite large. The threshold parameters has a value of 2^{15} as a default, lets us then consider matrix-matrix multiplication again (Figure 4). Here we have 4 thresholds, which would leave $(2^{15})^4 \approx 1.153 \times 10^{18}$ different threshold parameter settings (and this is assuming that 2^{15} is an appropriate maximum value for all hardware, which is not the case). However we see that a setting can only end in five different executions. This is due to a data value of 128 would result in the same in all the settings from 2^{15} down 128. So to manually tune the search space is incredible tedious and repetitive.

Portability of code is important, especially for GPU's due to the low level link between the code and the architecture (GPU threads, blocks/groups etc.). When tuning we tune to specific hardware, and representative datasets, so as soon as the code is to be executed on a different system (with a potentially different size of input data), the tuning performed would be useless (and in the worst case even detrimental to performance).

To illustrate the problem of repetitiveness through the search space we see Figure 4, with the thresholds $t = [10, 20, 30, 40]$, and the dataset values $dVal = [20, 30, 40, 50]$, so the predicate of $t_0 \leq dVal_0$ would result in taking the **true** branch, however this is also the case for $t_0 = 11$, making these two different settings, having the same dynamic behavior.

References

- [1] Guy E. Blelloch and Gary W. Sabot. “Compiling Collection-oriented Languages Onto Massively Parallel Computers”. In: *J. Parallel Distrib. Comput.* 8.2 (Feb. 1990), pp. 119–134. ISSN: 0743-7315. DOI: 10.1016/0743-7315(90)90087-6. URL: [http://dx.doi.org/10.1016/0743-7315\(90\)90087-6](http://dx.doi.org/10.1016/0743-7315(90)90087-6).
- [2] Guy E. Blelloch et al. “Implementation of a Portable Nested Data-parallel Language”. In: *SIGPLAN Not.* 28.7 (July 1993), pp. 102–111. ISSN: 0362-1340. DOI: 10.1145/173284.155343. URL: <http://doi.acm.org/10.1145/173284.155343>.
- [3] Troels Henriksen. *Streaming Combinators and Extracting Flat Parallelism*. June 2019. URL: <https://futhark-lang.org/blog/2017-06-25-futhark-at-pldi.html>.
- [4] Troels Henriksen. *Why Futhark?* May 2019. URL: <https://futhark-lang.org/>.
- [5] Troels Henriksen et al. *Experimental infrastructure for the paper “Incremental Flattening for Nested Data Parallelism” at PPOPP’19*. May 2019. URL: <https://github.com/diku-dk/futhark-ppopp19>.
- [6] Troels Henriksen et al. “Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates”. In: *SIGPLAN Not.* 52.6 (June 2017), pp. 556–571. ISSN: 0362-1340. DOI: 10.1145/3140587.3062354. URL: <http://doi.acm.org/10.1145/3140587.3062354>.
- [7] Troels Henriksen et al. “Incremental Flattening for Nested Data Parallelism”. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPOPP ’19. Washington, District of Columbia: ACM, 2019, pp. 53–67. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295707. URL: <http://doi.acm.org/10.1145/3293883.3295707>.
- [8] Rasmus Wriedt Larsen and Troels Henriksen. “Strategies for Regular Segmented Reductions on GPU”. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. Oxford, UK: ACM, 2017, pp. 42–52. ISBN: 978-1-4503-5181-2. DOI: 10.1145/3122948.3122952. URL: <http://doi.acm.org/10.1145/3122948.3122952>.
- [9] nVidia. “NVIDIA CUDA Compute Unified Device Architecture, Programming Guide”. In: (2008). URL: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.

A Code examples

A.1 Matrix Multiplication - CUDA for GPUs [9, p. 71]

```
1 __global__ void Muld(float* A, float* B, int wA, int wB, float* C)
2 {
3     int bx = blockIdx.x;
4     int by = blockIdx.y;
5     int tx = threadIdx.x;
6     int ty = threadIdx.y;
7
8     int aBegin = wA * BLOCK_SIZE * by;
9     int aEnd   = aBegin + wA - 1;
10    int aStep  = BLOCK_SIZE;
11    int bBegin = BLOCK_SIZE * bx;
12    int bStep  = BLOCK_SIZE * wB;
13
14    float Csub = 0;
15    for (int a = aBegin, b = bBegin;
16         a <= aEnd;
17         a += aStep,
18         b += bStep) {
19
20        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
21        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
22
23        As[ty][tx] = A[a + wA * ty + tx];
24        Bs[ty][tx] = B[b + wB * ty + tx];
25
26        __syncthreads();
27
28        for (int k = 0; k < BLOCK_SIZE; ++k)
29            Csub += As[ty][k] * Bs[k][tx];
30    __syncthreads();
31 }
32
33 int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
34 C[c + wB * ty + tx] = Csub;
35 }
```