Thesis description for Auto tuning Futhark

Simon Rotendahl and Carl Mathias Graae Larsen
{*simon, cala*}di.ku.dk

1 Problem

In this project we want to improve the current autotuner for the Futhark programming language. Futhark is a purly functional programming language created at DIKU that attempts to solve the problem that programming for GPUs is currently done at a low level with tools like CUDA or OpenCL. Futhark does this by abstracting away most of the problems with writing code for GPUs, and having a compiler that can heavily optimize the code. One such optimization is the amount of parallelism needed to probably saturate the hardware.

1.1 Parallelism in Futhark

When a Futhark program is compiled, it creates different versions of the code, take for example matrix-matrix multiplication, see Figure 1 for an implimentation 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
```

Figure 1: An implimentation of matrix multiplication in Futhark

When the Futhark compiler compiles this program it will generate three different, but semantically equivalent, versions of the program, each exploiting a different amount of parallelism. The three versions are:

1. A version where each map is run in parallel, but the *map-reduce* is run sequentially. This will launch $n \times p$, OpenCL-workitems. This represents a scenario where the number n and p are relatively small numbers.
2. If the matrices are small enough to allow the inner *map-reduces* to be fit in a single OpenCL-workgroup (The size of the workgroups are determined by the hardware the program is running on) a single workgroup can be launched for each of the *map-reduces*, meaning we end up with $n \times p \times m$ work-items.
3. If the matrices are too big to allow *map-reduce* to fit in one work-group, the *map-reduce* needs to be segmented. This version also launches $n \times p \times m$ OpenCL-workitems.

The program will then dynamically choose which version is used at runtime.

1.2 Choosing the optimal code version

A compiled Futhark program has multiple versions of the code, like the matrix-matrix (MM) example mentioned in section 1.1. Now image that it is known that every matrix given to the MM, is square and has a dimensionality less than 3. In this case it is slower on the GPU, than a sequential run on the CPU, due to overhead, mainly data transfer from system memory to the video memory on the GPU, and back again.

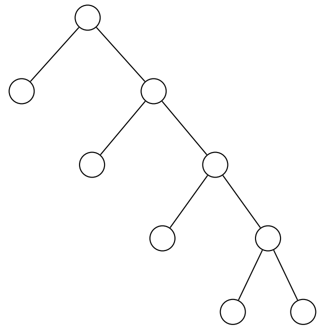
Figure 2: Example of choices through a Futhark program

Therefore it will not be optimal to always chose the highest level of parallelism possible. These choices in the program form an unbalanced tree structure, and example can be seen in figure 2.

Unbalanced The tree is unbalanced because if a comparison is true, the hardware is saturated, and we should not increase the amount of parallelism, and going right represent further parallelism.

Parameters As each of these nodes represent a choice of sequential vs. parallel code versions, there need to be someway to make that choice. Currently this is a less than comparison, between an integer parameter, determined during compilation, and a constant value (it might be possible that other expressions are better to use, than the comparison, but this is out of scope of the project, see 4). To get the optimal path through the tree, the constant values need to be chosen, so the hardware is utilized to the fullest. The path ends in a leaf node, and this leaf node represent an optimal version of the program, based on the data set given.

2 Autotuning

Autotuning is the process of automatically picking values for comparison that, given a representative set of datasets, optimizes hardware utilization, by picking the fastest version of the code, generated by Futhark.

Autotuning is helpful for two main reasons 1) The number of parameters might be big, and the values that they can take on is in a large range, thereby creating a vast search space, this space can very well be unfeasible for a human to look through 2) A parameter value could result in a OpenCL-workgroup size of x being chosen, but this optimal workgroup would be different from GPU to GPU. Autotuning solves these issues, a computer can search through a vastly greater space, than a human can, and an autotuning makes for these optimations to be portable.

Getting back to the search space. As an example, we might have a program with 5 parameters, with values ranging from 0 to 100,000 (there needs to be some range, otherwise the search space will be infinite, 0 to 100,000 is choosen abitraly for this example). This leaves the following search space

$$SearchSpace = \{(t_0, t_1, t_2, t_3, t_4) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}\} \quad t_i \leq 100,000$$

Where the *t's* is the parameters of the program, and the tuple represent a configuration of these parameters, that will result in some version of the code being chosen. This results in $5^{100,000} \approx 1 \times 10^{69,897}$ configurations. For comparison it is believed that there is 10^{82} atoms in the observable universe.

2.1 Important observations

There are two observations to make when tuning a Futhark program. The first is that different parameters might be encountered multiple times, due to functions being execute multiple times. Consider the right child of the root node in figure 2. This node might represent matrix-matrix multiplication. The first time it is executed, it might result in true (meaning optimal for nested parallelism) and takes the right child node, the next time it is executed it might be with a smaller array, which is not optimal for parallelism and should be executed sequentially, but the constant is already set, and the program will chose to run it parallel.

The second is that some of the parameters might be dependent on each other. An example of this is if making one decision early in the tree always means that we go down a specific path at some point later. This would mean that the second parameter is dependent on the first. Because of the dependency we could now simply look at the first parameter, and ignore the second when tuning the program.

2.2 The Current Autotuner

The current autotuner does not currently take into account the fact that Futhark programs are mostly small. The reason for this is that Futhark is intended to be Incorporated into a larger program, that is written in another language (C, Python, or F#), so that the parts of the program that benefits from running on graphics card, can be executed as such, through Futhark. The current autotuner is implemented with OpenTuner (see 6), which uses search techniques that are efficient when the search space is very large. Since the programs are small, the search space will be so as well, which means that it might be possible, and even faster, to search through the entire search space, thereby guaranteeing the optimal parameters.

3 Motivation

It is of great importance to utilize GPGPU's to their fullest potential, since the theoretical computational performance, is vastly greater, than CPU's. Since the current autotuner can sometimes spend a very long time tuning certain Futhark programs, and you would preferably be able to run the autotuner each time you've written a new program, it's currently not ideal.

4 Scope

The scope of this project is to tune the parameters that determine the path (the optimal level of parallelism.) through a compiled Futhark program. We will not be looking at tuning OpenCL-parameters and compiler flags. We will also not be analysing wheather expressions are better at making optimal chioces, than the comparison.

5 Learning Goals

- Learn when to choose different levels of parallelism by creating an autotuner to do just that.
- Gain a deeper understanding of how Futhark uses threshold parameters to choose the execution path depending on the input,
- and how to utilize the understanding of Futhark, and how execution paths are chosen, to limit the search space of possible paths.
- Acquire a better understanding of how to efficiently search through a search space for the optimal values.

6 Refrences

- Frederik Thorøe: Auto-tuning of threshold-parameters in Futhark. June, 2018. (pdf)
- OpenTuner (Website)