DEPARTMENT OF COMPUTER SCIENCE AT THE UNIVERSITY OF COPENHAGEN

Bachelors project

Auto-tuning Futhark

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

Contents

May 2019

Abstract

1 Introduction

2 Background

2.1 Futhark

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 wast amount of work has gone into transforming high-level hardware-agnostic code into these low-level GPU-specific languages [4].

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!"* [2]. 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 [3]

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 to 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.

2.2 Flattening

It is difficult to exploit nested data-parallelism. An approach to this problem is flattening [1]. The aim is to transform nested parallelism, into one-level parallelism. To briefly understand the concept of flattening, we will flatten a nested collection; let `A = [1, 2, 3, 4, 5, 6]`, this can be flattened into what is called a **field** [1], here represented as a tuple, but it could be a different data-structure. The first element, of the field, is a collection representing the length of the segments, and the second is either a collection, or another field, giving; `field = (shape, val)`. The collection `A`, flattened into a field, would be `(Ashape = [6], Aval = [1, 2, 3, 4, 5, 6])`. This technique can be applied recursively to any depth. A depth of `n` would give `n` shape collections, for example let `B = [[1, 2], [3, 4, 5], [6]]`, this would be flattened to the field `B = ([3], (Bshape, Bval))` $\rightarrow$ `B = ([3], ([2, 3, 1], [1, 2, 3, 4, 5, 6]))`.

With the collection flattened to a field structure, work can now be applied in parallel, by stepping up or down in the field, for example taking the head of each nested collection would step down in `B` once, to get `step-down([3], ([3], ([2, 3, 1], [1, 2, 3, 4, 5, 6]))) = ([2, 3, 1], [1, 2, 3, 4, 5, 6])`, and then distributing the head function to each segment in parallel, by giving each segment of `[1..6]` to a core and performing the function, giving back the field `([3], [1, 3, 6])`.

With the ability to flatten, nested parallelism can be mapped to hardware, as one-level. However this is not necessarily optimal, as it does not take the capability of the hardware into account, leading to poor utilization of locality optimization. The problem then becomes how to optimize for nested parallelism, hardware capabilities, and the data being worked on.

Going back to the matrix-matrix multiplication example in listing 1, there are 3 level of nested parallelism to exploit the outer and inner `map` on line 5, and the `redomap` in the `dotprod` function. With these three levels there is, at least, 3 different ways to execute the code [4]; **1)** If the size of the two maps is not big enough to saturate the hardware, then the `redomap` map should also be executed in parallel. **2)** Otherwise the `redomap` should be sequentialized, to optimize locality. **3)** If the parallelism in the two `maps`, would over-saturate the hardware, then another chunk can be sequentialized, to optimize locality.

The optimal choice here will, as stated, depend on the hardware, and the data being worked on.

2.3 Incremental flattening

Futhark has taken an approach to this choice, mentioned in the section above, where it generates

several piecewise, semantically equivalent, code versions of the same code in a program, each exploiting a different level of parallelism, which are then discriminated at runtime by statically generated predicates [4].

The process of picking a code version is done using the generated predicates, and *threshold parameters*, and is called tuning, we tune the parameters to get the best runtime based on the hardware, and data given. Currently the predicates are a simple less-than comparison, but that could change. By default the threshold parameter is set to a value off 2^{15}, as an estimate, but is likely sub-optimal. Due to the way the code versions are generated, they can be dependent on each other. If we think back to the example in listing 1, we saw that there were, at least, 3 different executions regarding exploitation of parallelism, this means that there is also 3 code versions, guarded by predicates. If the predicate for the two outer `maps` shows they would saturate the GPU, then the code version where the parallelism would over-saturate the GPU, and run the entire function sequentially, would not get picked, we the step down into the `redomap`, here there is a predicate that decides if `redomap` should also be executed parallel, or whether it would over-saturate, and be executed sequentially. From this it is clear that the predicates/threshold are dependent on each other.

To inspect the structure of these predicates and thresholds parameters further, lets look at an actual Futhark program;

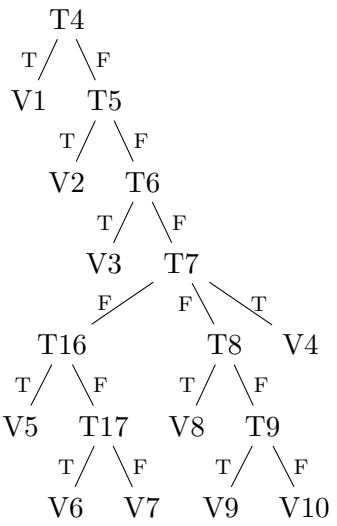

Figure 1: The dependencies between thresholds, of the test program `LocVolCalib.fut`. The dependency is based on a comparison of the threshold (T), the edge to the next node is then taken based on that comparison. Each path through the tree, to one, or more, code versions (V), is then an execution of the program.

To tune a program we need to examine each execution path through the tree. It is important not to get an end node confused with an execution path. Two example of paths through the tree in figure 1, that show this, could be;

- {`(T4, False), (T5, False), (T6, False), (T7, True)`}
- {`(T4, False), (T5, False), (T6, False), (T7, False), (T8, False), (T9, True), (T16, False), (T17, 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 is 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 independent code versions, that are to be combined.

3 Design

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.

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

[3] 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.

[4] 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.

[5] 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 [5, p. 71]

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

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

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

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

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

        __syncthreads();

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

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