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

Stuff

1 Introduction

Stuff

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:

```
let dotprod [n] (xs: [n]f32) (ys: [n]f32): f32 =
    reduce (+) 0f32 (map2 (*) xs ys)

let main [n][m][p] (xss: [n][m]f32) (yss: [m][p]
    f32): [n][p]f32 =
    map (\xs -> map (dotprod xs) (transpose yss))
    xss
```

Figure 1: Matrix-matrix multiplication in Futhark [3]

A Futhark program for matrix-matrix multiplication can be seen in figure 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])`.

2.3 Incremental flattening

A principal critical for this code transformation from low-level GPU languages to Futhark, is flattening.

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;
}
```