Hardware-oriented optimisation in CUDA

University of Regensburg

Tobias Sizmann

day month year

Abstract

Contents

1 Introduction **4**

1.1 From single-core CPUs over multi-core CPUs to GPUs 4

1.2 What is CUDA 4

2 Tree Reduction on GPUs **7**

2.1 The importance of reductions 7

2.2 The tree reduction algorithm 7

2.3 Naive implementation with CUDA 8

2.3.1 Serial tree reduction on a CPU 8

A Appendix One **11**

Chapter 1

Introduction

1.1 From single-core CPUs over multi-core CPUs to GPUs

With the invention of the MOS transistor in 1959 and more specifically the silicon-gate MOS transistor in 1968 the development of single-chip microprocessors started. The first processors were single-threaded and their bottleneck was (besides other things) their clock rate. However, clock rates increased exponentially over time and subsequentially did the computing power. In the 2000s this development was eventually brought to a halt at around 3.4 GHz due to thermodynamical limits of silicon. While pushing past that limit is possible, the extra costs for cooling usually outweight the increase in computing power. This is the beginning of the multiprocessing era. The idea is simple: When one thread cannot run faster, just increase the number of threads. In 2006 the first desktop PCs with two cores were sold and ever since the number of threads is steadily increasing. There was one problem in particular, however, that CPUs (Central Processing Unit) were not efficient in solving, namely, rendering graphics. This task required a lot of independent and small calculations that needed to be done in real time - a prime example for a massively parallelisable computation. Since rendering graphics required such a different type of computing, the first GPUs (Graphics Processing Unit) were developed. These GPUs featured less single thread computation power but have a higher number of threads. State of the art GPUs have thousands of threads. This being said, the number of threads cannot be easily compared between a CPU and a GPU or even between different GPUs as they follow different design paradigms. How the GPU threads work exactly will be explored in this work. Generally, one can say that GPUs perform well in massively parallelisable computations where the single computations are not complex while CPUs excel at complicated single threaded problems (for example running the event loop of a desktop application). This shift to a higher number of threads rather than thread quality has been greatly motivated by scientific calculations, machine learning and graphics.

1.2 What is CUDA

Writing code for a GPU is not as straight forward as for a CPU since it is more dependent on its hardware. Different GPU developers use different application pro-

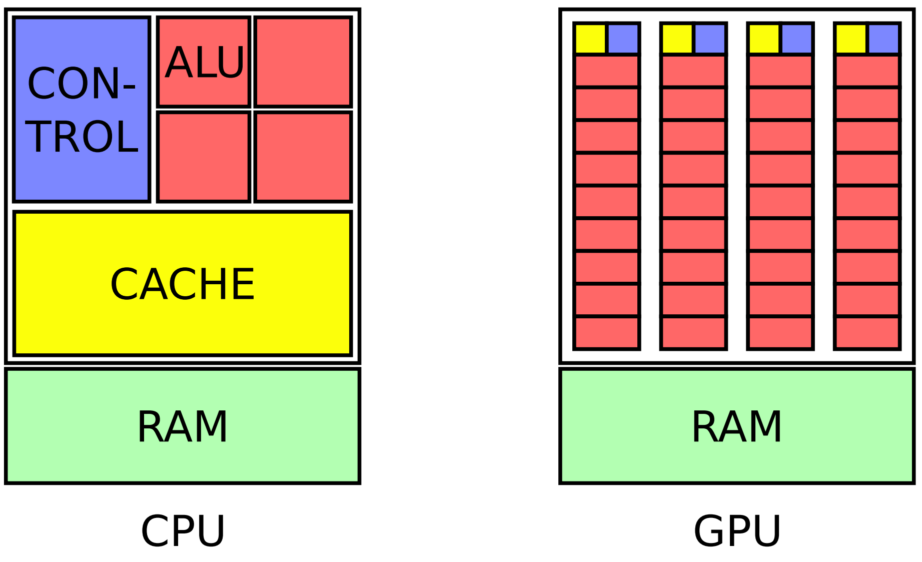

Figure 1.1: Comparison of the basic architectural differences of a GPU and a CPU. In green the random access memory (RAM), in yellow the cache, in blue the flow control unit and in red the arithmetic-logic unit (ALU). Both designs feature a RAM that all threads have access to. The GPU is split into many small "CPUs" (vertical groups) with their own flow control and cache. These are called streaming multiprocessors (SMPs or SMs). Generally the cache hierachry is more complicated as depicted in the figure. However, the defining property here is that there exists a non-global cache level that is assigned to a group of ALUs, namely the SMs. Note that the notion of an ALU has slightly different meaning for a CPU and a GPU. For a CPU one ALU usually corresponds to one thread. For a GPU one ALU usually corresponds to a group of threads (for modern GPUs: 32), called a warp.

gramming interfaces (APIs). The biggest GPU designer Nvidia developed a framework called CUDA, which will be used in this work. It is simple to learn, offers efficient implementations and a plethora of literature and support. The downside is that it only supports Nvidia GPUs and is not open-source. An alternative would be OpenCL, which is open-source but harder to learn.

CUDA works as an extention to the C programming language and comes with its own compiler. Sections of code are distributed to either the CPU (called host) or the GPU (called device). When only writing code for the host normal C is used. Writing device code is more sophisticated. Here, the programmer first needs to write a so called kernel, which can be thought of the interior of a for-loop. Then the host must transfer required data to the memory of the GPU and call the kernel. When calling the kernel, the boundaries of the for-loop are set. The loop is then executed in parallel on the GPU. This API only allows parallelisation of for-loops. While this might seem like a very strict limitation, it closely relates to how a Nvidia GPU works.

The GPU can be thought of being organised in streaming multiprocessors (SMPs or SMs), warps and threads. Warps will be explained in more detail later. A SM groups together a set of (hardware) threads and allows synchronisation between them. Threads of different SMs cannot be synchronised. Also, threads within a SM share a cache, which cannot be accessed by threads of another SM. The domain of the for-loop is organised in blocks and (CUDA) threads, where the threads are grouped in blocks. The blocks are executed by the SMs, which means that only threads within a block can be synchronised and have access to the same cache. Note the differentiation of hardware and CUDA threads. This is an unfortunate ambiguity in the terminology of CUDA. While each CUDA thread is mapped to one hardware thread, a hardware thread can be mapped to (i.e. execute) several, none or one CUDA threads. This will become clearer in the next chapter.

Chapter 2

Tree Reduction on GPUs

2.1 The importance of reductions

A reduction in terms of parallel programming is an operation, where a large dataset of entries (e.g. numbers) is reducted to one entry. The simplest example is the sum of a set of numbers and will be used for the rest of this work. More generally are reduction is defined by an operation $\circ : X \times X \to X$ on two entries with the following properties:

$$a \circ b = b \circ a \quad \text{(commutativity)}, \tag{2.1}$$
$$a \circ (b \circ c) = (a \circ b) \circ c \quad \text{(associativity)}. \tag{2.2}$$

This guarantees that the result is (mathematically) independent of the order in which the elements are reduced. Note, that these properties do not ensure numerical stablility.

The reduction operation, first and foremost the sum, plays a crucial role in all of numerics. Simple linear algebra operations like the scalar product or the matrix multiplication already include a reduction: The entries of two vectors are multiplied element wise and the summed up. In machine learning, reductions are present as a key step in feed-forward networks: Again all the values of the node one layer below are multiplied by weights elementwise and the summed up to calculated the value of a singular node above. Reductions are a very basic and fundamental operation and, therefore, an efficient implementation is required.

2.2 The tree reduction algorithm

The most efficient algorithm is heavily dependent on the underlying hardware. For example, a node network with a lot of parallelisation overhead and a complicated communication topology might use a ring algorithm (”add value and pass to next”). In this work the focus is on single GPU reductions. Here, the tree reduction approach is the most successful. All available threads are called to reduce two entries to one in parallel, effectively halfing the size of the dataset in one step. This is repeated until only one entry remains.

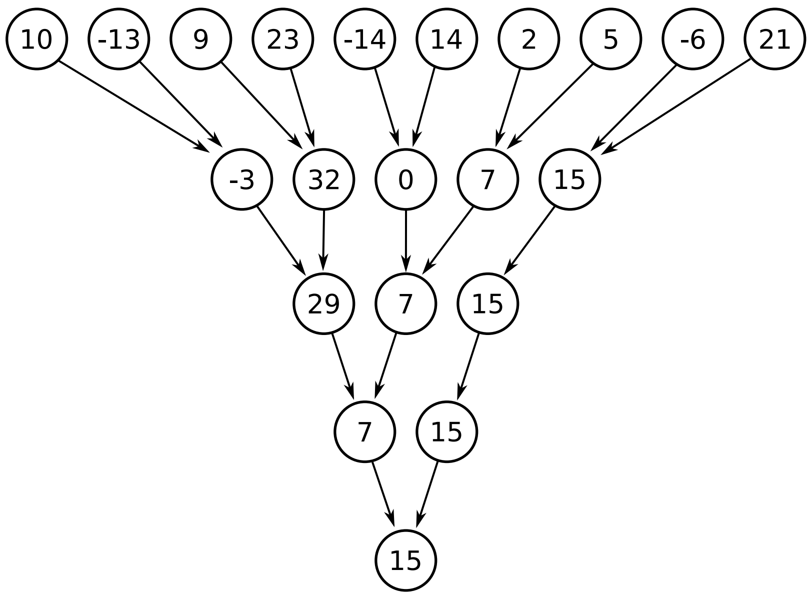

Figure 2.1: Example of a tree reduction of ten elements. Note how there is a leftover after the second reduction. These are usually handled by zeropadding (i.e. adding zeros) after each step to make the number of elements divisible by 2.

2.3 Naive implementation with CUDA

For a better understanding of the framework of CUDA and tree reduction algorithm an unoptimized code is presented which implements the algorithm for the case of addition of signed 32-bit integers. Note, that even though this is unoptimized GPU code, it runs magnitudes faster than on a CPU (exact speedup depending on the problem size and the available hardware).

When approaching a problem using CUDA the big challenge is to map the for-loop that one wants to parallelize to blocks and CUDA-threads. Often, the best starting point is to write serial CPU code to get a feeling for the problem and to have a working solution to test the optimized solutions against later. In this case, we will first implement a serial tree reduction in C and port this code to CUDA in a second step.

2.3.1 Serial tree reduction on a CPU

Our starting point is an integer array `h_in`. The `h_` denotes data stored on the host (CPU RAM), which is a useful convention, since CUDA does not distinguish between pointers to host and device memory. For simplicity, we assume that the size of the array is a power of two. If this is not the case, one could simply zeropad the data and would still be able to use the code.

Lets write a function `reduce` that executes the tree reduction. This function takes an input array `h_in`, performs the reduction and writes the result to an output `h_out`. We need two for-loops: One to iterate over the step of the reduction (vertically in fig. 1.1) and one to iterate over the (remaining) data set (horizontally). A temporary array is used to do the reduction on, so the input array stays untouched.

```c
void reduce(int * h_in, int * h_out, int len) {
    // copy data into temporary array
    int * temp = (int *) malloc(len*sizeof(int));
    memcpy(temp, h_in, len*sizeof(int));

    int stride = 1;
    // iterate over reduction steps (vertically)
    for (int step = 0; step < log2(len); step++) {

        // iterate over dataset (horizontally)
        for (int i = 0; i < len; len += 2*stride) {
            temp[i] = temp[i] + temp[i + stride];
        }
        stride *= 2;
    }

    // return the result
    *h_out = temp[0];
    return;
}
```

The inner loop of this implementation could be parallelised easily, since each step of the inner loop is independent. This means, however, that with each step of the outer loop all threads are created and destroyed, which creates a large parallelisation overhead. In general and independent of CPU or GPU, one should always aim to parallelize the outermost loop. Here, the outer loop cannot be parallelized, since each step is dependent on the result of the step before. To fix this, one can swap the two loops. While this requires slightly more code, it greatly simplifies all examples in the following. The rewritten function looks like this:

```c
void reduce(int * h_in, int * h_out, int len) {
    // copy data into temporary array
    int * temp = (int *) malloc(len*sizeof(int));
    memcpy(temp, h_in, len*sizeof(int));

    // iterate over dataset (horizontally)
    for (int i = 0; i < len; len += 1) {
        int stride = 1;

        // iterate over reduction steps (vertically)
        for (int step = 0; step < log2(len); step++) {
            if (i % (2 * stride) == 0) {
                temp[i] = temp[i] + temp[i + stride];
            }
        }

        stride *= 2;
    }

    // return the result
    *h_out = temp[0];
    return;
}
```

Note that an additional if-statement is required. This is a much better basis for parallelisation, since now the outer loop can be parallelised. However, one needs to be very careful as this is now prone to a race-condition.

It should be noted, that there are much faster ways to implement a reduction algorithm as a single-thread CPU application. This code has been written with the intent of parallelisation on a GPU and serves solely this purpose.

Appendix A

Appendix One

this will be space for the appendix