Chapter 1

Introduction

Up until halfway the first decade of the new millennium it was possible to gain computing performance whilst also being able to maintain the sequential programming paradigm. This was due to Moore's law, stating that the number of transistors on integrated circuits double approximately every two years. There was no need for research into explicit parallelism because the next generation of computing devices was just around the corner which would make the research obsolete.To perpetuate the sequential programming paradigm several innovations such as multiple issue, deep pipelines and out of order execution were introduced into processors which were inefficient in both the use of transistors and power. Eventually though it became impossible to progress any further whilst still supporting the sequential paradigm. The integrated circuit industry was unable to continue decreasing the size of MOSFETs whilst continuing to increase the clock frequency. The industry had hit what is called the power wall. The solution to this problem was to go over to parallel processors, meaning that there is more than one processing unit working at a time. A lot of real world applications are parallel, and hardware can be made parallel with relative ease. The problem lies in the programming model, how to exploit this parallelism and make programming for these parallel architectures easier and transparent for the programmer.

1.1 Computing Components

1.1.1 Multicore Processor

The multicore processor is the solution presented for the aforementioned problems by the traditional CPU manufacturers such as Intel and AMD. The idea behind this type of processor is to place a number of cores (currently up to eight) on the same die. This presents a compromise between maintaining sequential perfor-

mance whilst also providing a certain advantage of parallel processing. Parallel programming for these processors presents certain challenges whilst their modest parallelism cannot provide a dramatic improvement in power performance. Multicore processors are unlikely to be a one-size-fits-all solution to the parallel problem.[3]

1.1.2 Graphics Processing Units

Graphics Processing Units Graphics processing units are a type of coprocessor in in traditional computers meant to process images for output to the display. Recently however there has been increased interest in the GPGPU, the general purpose graphics processing unit. These processors implement a different paradigm, namely the manycore paradigm. A GPU is a processor with hundreds single instruction multiple data cores, each of which is heavily multi-threaded. Because of this large amount of cores the FLOPS (floating point operations per second) is unrivalled. [4]GPU's, due to their SIMD nature present some problems, conditional execution paths for example, present a serious overhead on the GPU. GPU's are programmed with either OpenCL (open standard) or CUDA (proprietary to Nvidia)

1.1.3 Field Programmable Gate Array

FPGAs are devices containing a vast amount of configurable logic linked by programmable connections. This logic is comprised of lookup tables grouped together into configurable logic blocks. Any combinatorial function can be programmed into these LUT's. Next to these uncommitted logic blocks a typical FPGA also contains several blocks with a specific function such as block ram and DSP multipliers. FPGAs are an interesting competitor in the parallel processing field because they aren't constrained by the Von Neuman architecture. FPGAs follow the dataflow paradigm in which the data flows through the logic. Implementing a data-flow is inherently parallel. The different stages in the datapath can also be made sequential effectively making the datapath a pipeline. The fine grained nature of FPGAs also means that the bitwidth can be adapted to the application.

1.2 Berkeley Dwarves

Image processing algorithms are very compute intensive. These makes them prime targets for exploiting parallelism and implementing them on parallel architectures. Which platform is the best fit however is dependent on both the

algorithm and the data. A common method to subdivide parallel algorithms is presented in [3], the so called ”dwarfs”. These 13 dwarves are classes of algorithms in which the membership is defined by a similarity in computation and data movement.These 13 dwarfs are classes of algorithms in which the membership is defined by a similarity in computation and data movement. The dwarfs are:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

A thorough review of these dwarfs and what kind of computation and communication they entail goes beyond the scope of this document. More information can be found on the Berkeley View Wiki [5] and an updated view can be found in [2]. Finding out which dwarf is most suited for which platform is a very labour-intensive task. In [5] a theoretical analysis of dwarf performance on different accelerators in heterogeneous systems is given. A first point to note is that for floating point operations GPU's are hard to beat. Fixed point numbers are a way to overcome this problem. Another point to note is that conditional elements and costly communication can wreak havoc on the accelerator's performance.

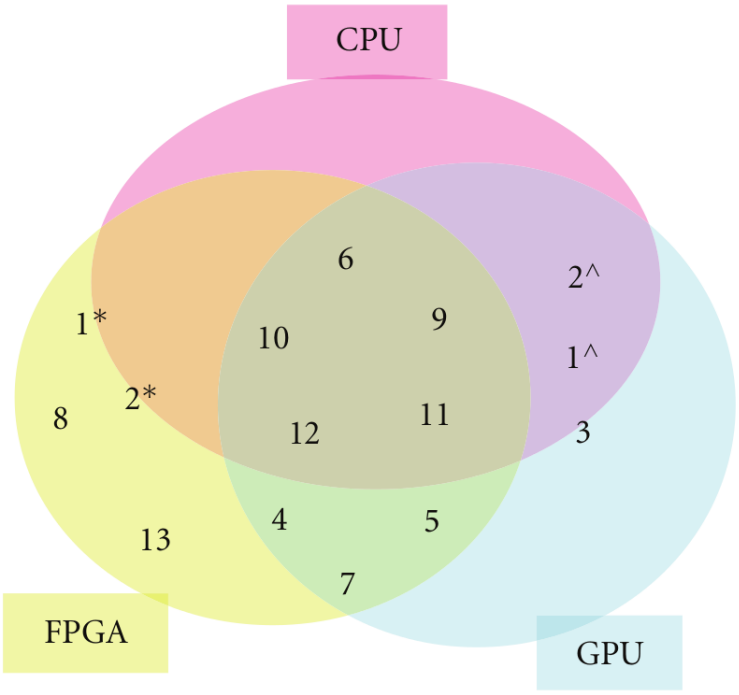

Figure 1.1: Venn diagram showing an analysis of different hardware accelerators in regards to performance on a certain dwarf. * denotes fixed point whilst ^ denotes floating point.[5]