Studies of classical HPC problems on fine-grained and massively parallel computing environment based on reconfigurable hardware.

Evaluation of an FPGA-based supercomputing platform

L A U R E N Z   C H R I S T I A N   B U R I

Master of Science Thesis
Stockholm, Sweden 2006

ICT/LECS-2006-88

Studies of classical HPC problems on fine-grained and massively parallel computing environment based on reconfigurable hardware.

Evaluation of an FPGA-based supercomputing platform

L A U R E N Z   C H R I S T I A N   B U R I

Supervisor
Olle Raab
Mitrionics AB, Lund, Sweden

Examiner
Vladimir Vlassov
Department of Microelectronics and
Information Technology IMIT
KTH, Stockholm, Sweden

Master of Science Thesis
Stockholm, Sweden 2006

ICT/LECS-2006-88

Abstract

Today, High Performance Computing (HPC) problems occur in various lines of business. Whilst conventional von Neumann processors are slowly approaching the maximum of feasible CPU frequency, become FPGA's an interesting alternative as they get large enough to be able to implement critical algorithms efficiently.

This thesis consists of an evaluation of the capabilities of the FPGA based supercomputing platform Mitrion regarding mathematical functions, ranging from linear algebra to trigonometry. The evaluation illuminates the achievable speed-up as well as the programmability of the reconfigurable processor and compares these aspects with ANSI-C solutions destined for an ordinary x86 AMD 64 processor.

For a short summary of the overall results please refer direcly to chapter 10.

Keywords : Mitrion, HPC, FPGA

Acknowledgements

The realization of this Master thesis at Mitrionics was both fun and a highly valuable knowledge enrichment. It was very interesting to work with their intelligent product, which adds cutting edge technology to the supercomputing world.

I would like to thank everybody at Mitrionics, especially my supervisor Olle Raab, for having supported me in realizing this diploma work. I would also like to thank my examiner, Vlad Vlassov, for help with the report and administration. Last but not least I thank my opponent, Piotr Kundu, for useful inputs, which helped me to improve the quality of this report.

Contents

List of Figures

Chapter 1

Introduction

1.1 Preamble

In spite of the tremendous evolution in computer hardware claim critical applications always more and more computing power. Unfortunately can a system rarely exploit the peak performance of its CPU because of dependencies such as relatively slow memories and interconnections. In addition are the CPUs slowly approaching the maximum in terms of clock frequencies of what is physically feasible.

The trend to fill the gap between the speeds of memory and processor and to overcome the limits of the clock rate is to parallelize computing and to customize hardware.

In this thesis, the author is going to evaluate the platform Mitrion, which is a massively parallel virtual processor used to implement algorithms efficiently on reconfigurable hardware.

1.2 Project specification

The massively parallel virtual processor Mitrion is based on reconfigurable hardware and is said to be about 20 times faster than a CPU. An aim of this project was to evaluate the Mitrion processor regarding this statement.

The evaluation was to be done by implementing a representative range of mathematical functions, ranging from linear algebra to trigonometry. The functions were to be implemented in the languages Mitrion-C, which is used to configure the Mitrion virtual processor, and ANSI-C for performing a quantitative benchmark test with a sequential processor. The sequential processor is represented by an AMD 64 processor with a x86 architecture.

It is expected that the Mitrion processor keeps its promise of being in average 20 times faster than the sequential processor.

Furthermore was a qualitative analysis between the massively parallel processor (MPP) Mitrion and a sequential processor to be done. This included the discussion about what advantages or disadvantages there exist when programming the MPP or the sequential processor, what kinds of algorithms are more appropriate for what processor type and what kind of possibilities there are to improve the programmability of the MPP.

1.3 Mitrionics

The company Mitrionics AB was founded in the year 2000 and is located in Lund, Sweden. It is their goal to exploit the maximum performance of classical HPC problems on their reconfigurable platform Mitrion. Their product, the Mitrion virtual processor, is reconfigured for every application. The configuration is done with the Mitrion Software Development Kit (SDK), which uses it's own language Mitrion-C. Compiling a Mitrion-C program by means of the Mitrion SDK yields a processor configuration, which is customized for the specific algorithm.

The Mitrion-C language introduces a high abstraction level to hardware programming. It makes reconfigurable hardware accelerated computing available for scientists in various fields of business without having to acquire deep knowledge in hardware. The table below is taken from the Mitrion marketing brochure [3] and compares the Mitrion Platform with VHDL or ESL design tools.

The Mitrion Platform compared to VHDL or Electronic System Level (ESL) design tools:	**VHDL or ESL design tools**	**Mitrion**
Typical application speedup	10-30 times**	10-30 times**
Application development time	Months or years	Days or weeks
Program without hardware considerations	No	Yes
Easily move applications to new platforms or FPGAs	No	Yes
Built-in support for floating-point numbers	No	Yes
Automatically find and utilize parallelism in software	No	Yes
Graphical simulator lets you visualize parallelism of software and find performance bottlenecks	No	Yes
Code efficiency: simple example adding two vectors	about 400 lines	7 lines of code
**Up to 100 times for some applications		

Chapter 3 provides an introduction to the concepts of Mitrion.

1.4 Assumed prior knowledge

It is assumed, the reader is familiar with basics in computer science. However, no prior knowledge in parallel computing or FPGA technology is required as short introductions in these topics are provided.

1.5 Reading directions

This document is structured as follows. The next chapter outlines foundations of parallel computing. Chapter 3 introduces the Mitrion platform. The chapters 4, 5, 6, 7 and 8 illustrate how a set of algorithms was implemented on the Mitrion platform. The illuminations include analysis of the algorithms, implementation design and performance evaluation. Chapter 9 consists of an overall discussion about the Mitrion platform followed by the conclusions in chapter 10. The last chapter 11 is about future work.

1.6 Related work

Prior to this diploma work, Johan Rees and Henrik Abelsson wrote closely related Master Thesis at Mitrionics AB in Lund. Johan Rees analyzed an intersecting set of functions as is discussed in this report, the results however differ.

The other student, Henrik Abelsson, wrote a similar thesis but analyzed a different set of algorithms. There was no collaboration between the writer and the mentioned students.

1.7 High Performance Computing

Today, the development of fast digital computers have caused traditional analytical calculations as well as experiments in various fields like fluid dynamics, electrical engineering or quantum chemistry to be replaced by computer simulations [4]. In that such problems are of extremely complex nature, it is often not feasible to calculate exact solutions on today's digital machines in full precision and within a reasonable amount of time. The problems have to be simplified and approximated. Often a trade-off between accuracy and computation time has to be made.

Such problems are of the High Performance Computing (HPC) domain and belong to the open problems in computer science [5].

1.8 The von Neumann machine

The digital computer model that is referred to as the *von Neumann machine* was contrived by John von Neumann around the year 1945 [6] and is defined as follows:

"The von Neumann machine is a stored program computer. It keeps its specific instructions (programs) in its memories, storing the information in the same manner as it stores any other information (data). The computer does necessarily contain five basic components: a control unit, memory, a calculating unit (CPU) and input and output for interacting with human users. The control unit delves into memory, finding an instruction or a piece of data, and deals with what it found accordingly." [6]

This model coincides still with most of the today's computers.

1.9 Limitations of the von Neumann model

Despite the tremendous CPU advances, memory bandwidth can not keep pace with the improvements in processor performance.

Today, the most important factor for computing speed is not how fast a processor can operate, much more does it depend on how fast data can be moved from one place to the other, i.e. how fast the memory can provide the CPU with data. The time needed by the memory system to provide a word of data requested by the CPU is referred to as *latency*. Another notation in this context is the *bandwidth*, which specifies the rate at which data can be transferred from the memory to the processor. It happens that a processor stalls for hundreds of cycles while waiting for data.

Caches constitute a good countermeasure to treat this problem. Basically, they are small memories embedded in the processor that contain copies of data of the main memory. Every time the processor makes a request to the memory, the cache provides the data in case it has a valid copy. Because caches are inbuilt in the processor, the latency is extremely low and almost negligible compared to the latency of the main memory. Several policies exist that define how data is copied to and replaced in the caches. Such policies become extremely important and have a crucial impact on performance in shared memory architectures where multiple CPUs have own copies of the memory that is shared among all processors of the system.

1.10 Benchmarking

As computer systems are very advanced and as different vendors develop their own architectures, it is hard to compare the performance of different systems by simply looking at their specifications. It is therefore common to measure the relative performance between systems in terms of floating point operations performance or execution time for running a given specially designed, so-called, benchmarking program. In many cases the benchmarking programs have do be adapted to the underlying platform, they should however produce the same output.

One has to observe that not every system was designed to perform at its theoretical peak for a certain benchmark. The benchmarking results are specific for the given problem. Furthermore, when comparing CPUs, it is often crucial what other hardware components are involved. One important point are for example the memory's capacity and speed.

Benchmarking in this thesis was done by comparing execution times for the given algorithms between the Mitrion processor and a classical sequential processor with von Neumann architecture. For all algorithms, there was an equivalent ANSI-C program implemented in order to be able to compare the performance. The execution times of the Mitrion-C programs were measured in simulation mode because of the higher accuracy of the result. Measuring the execution times in the host program instead would have added very little overhead. If not otherwise stated, the benchmarking platform for the C programs was an AMD 64 3200+ processor that runs at 2000 MHz and has a 512KB L2 cache and 1024MB random access memory. The C programs were compiled on the same machine with Linux Ubuntu 5.10 and GCC 4.0.2 using the option -O3, which does some optimizations such as function in-lining.

It is known that GCC, which is an Open Source compiler, has less abilities to optimize programs for the underlying hardware than commercial compilers, which are specially designed for specific processors. GCC was chosen because it is widely-used and because it is probably the best C compiler that is available for free.

Moreover has the author to admit that it is not guaranteed that the operating system does not do any context switches during a single execution of a benchmarking program. Such interruptions can have impacts on the cache hit-rate and hence slow down the execution. To establish an estimate of the execution times of the C programs is difficult because of the role of the caches and was therefore left out. The execution times stated are averages over running the programs several times.

Chapter 2

Basic terms and concepts of parallel computing

This chapter illuminates some basic concepts and terms of parallel computing including complexity notations, data-parallelism and pipelining.

2.1 Motivation for parallel computing

The idea of parallelizing computing is not new and there exist several techniques to achieve different degrees of parallelism. The purpose to do so is to speed up computing by processing more data per time frame than the strictly sequential model.

2.2 Complexity notations

Oftentimes, there exist several algorithms that can solve a given problem. In order to be able to compare the potential performances of the contemplable algorithms, it is common to use the notation of the asymptotic growth of a procedure.

A function $f(x)$ is said to be of order $\theta(g(x))$ when the following holds:

$$f(x) \in \theta(g(x)) \Leftarrow c_1 g(x) \leq f(x) \leq c_2 g(x) \tag{2.1}$$

$f(x)$ is of order $O(g(x))$ if the equation below is true:

$$f(x) \in O(g(x)) \Leftarrow f(x) \leq cg(x) \tag{2.2}$$

Note that $f(x) = \theta(g(x))$ implies $f(x) = O(g(x))$. The θ notation is referred to as the exact magnitude whereas the big O notation is the upper bound of the algorithm.

2.3 Data dependency

Not every program or algorithm can be parallelized to any desired degree. The main reason is that there often exist data dependencies among different calculation steps. To exemplify, consider the two programs below.

```
Procedure vectorSum(int[n] a)
b = 0
for $i = 0$ to $n - 1$ do
    $b := a[i] + b$
end for
return $b$
```

and

```
Procedure addVectors(int[n] a, int[n] b)
for $i = 0$ to $n - 1$ do
    $c[i] := a[i] + b[i]$
end for
return $c$
```

To illustrate the above code snippets, their corresponding data dependency graphs are drawn in 2.1.

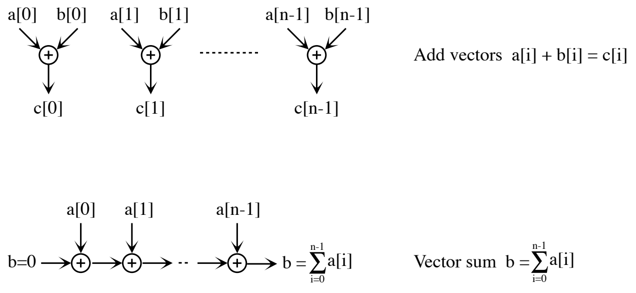
a[0] b[0]
a[1] b[1]
a[n-1] b[n-1]
c[0]
c[1]
c[n-1]
Add vectors a[i] + b[i] = c[i]
a[0]
a[1]
a[n-1]
b=0
b = $\sum_{i=0}^{n-1}$a[i]
Vector sum b = $\sum_{i=0}^{n-1}$a[i]

Figure 2.1: Data dependency graphs

It is clearly visible in the dependency graphs, that the sums in the procedure *addVectors* can entirely be executed in parallel while the sums in *vectorSum* must wait on a previous result.

2.3.1 Critical path

Assuming that every operation (circle with operation sign in 2.1) takes one unit of time (one clock cycle) the *critical path* is defined as the number of stages that are necessary to obtain the final result. In the schematic illustration 2.1 it is equal to the computation time or the latency for to obtain a result.

The procedure *addVectors* has a critical path length of 1 while the procedure *vectorSum* has a critical path of length n. Optimizations are possible to reduce the critical path for the procedure *vectorSum* to $\log n$. The summing would then have to be performed in a reversed tree (reduction tree) like fashion.

2.4 Speedup

When implementing a parallel version of a serial program, the programmer is often interested in how much execution time could be saved by the parallel program over the sequential version. Speedup, denoted by S, is defined as the relative benefit of solving a problem in parallel [7]. In terms of the asymptotic notation the formula is:

$$S = \theta\left(\frac{T_{sequential}}{T_{parallel}}\right) \tag{2.3}$$

where $T_{sequential}$ and $T_{parallel}$ are θ-complexity notations.

In practice, S is often expressed as the ratio of the serial run-time to the time taken by the parallel algorithm for solving the same problem ($S = \frac{T_{sequential}}{T_{parallel}}$). Despite the fact that the problem and the problem size have to be the same for both architectures, the algorithms to solve the given problem might be different since the execution times are measured for the best suited algorithms for the respective platforms.

2.4.1 Amdahl's law

Amdahl's law [8] states that the maximum achievable speedup of a program does not depend on the number of processing elements available, it is rather the algorithm itself that determines the upper bounded speedup. If $p \in [0, 1]$ is the fraction of the algorithm that can be parallelized with a speedup s_p the total speedup is given by the formula 2.4.

$$S_{tot} = \frac{1}{(1-p) + \frac{p}{s_p}} \tag{2.4}$$

Considering the speedup of the parallelizable part of the program to be extremely high ($s_p \to \infty$), the formula 2.4 grows asymptotically to:

$$S_{max} = \frac{1}{(1-p)} \tag{2.5}$$

Which means that the maximum speedup is always upper bounded by the sequential (not parallelizable) part of the algorithm.

2.5 Achieving parallelism

Basically two types of parallelism can be named; *data-parallelism* and *pipelining*. Both models are presented shortly.

2.5.1 Pipelining

Pipelining works actually the same way like a car assembly line. When one car can be built within D days from beginning to the end, can an assembly line have an output of a number of cars per day. This is achieved by manufacturing the final product in several stages, where every stage can be done independently and simultaneously. If the longest stage takes t units of time, the factory can sell a car every t units of time. If f is the fraction l/t, where l is the latency of the production (strictly serial production time), the productivity is f times higher over the strictly serial production. This principle is extremely efficient in terms of saving time and money.

Single processors use this technique to improve their efficiency like it does for example the Intel Pentium 4, which has a 20 stage pipeline [7]. This principle

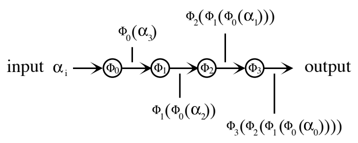
$\Phi_2(\Phi_1(\Phi_0(\alpha_1)))$
$\Phi_0(\alpha_3)$
input α_i
Φ_0
Φ_1
Φ_2
Φ_3
output
$\Phi_1(\Phi_0(\alpha_2))$
$\Phi_3(\Phi_2(\Phi_1(\Phi_0(\alpha_0))))$

Figure 2.2: Snapshot of a pipelining system

is also referred to as *implicit parallelism.* However, to implement long pipelines efficiently needs a good branch destination prediction technique since every 5^{th} to 6^{th} instruction is a branch instruction [7].

Applying pipelining to the previously presented *vectorSum* procedure would mean to subsequently calculate sums of different vectors.

As shown in picture 2.2, each processing element calculates one step Φ_i and transmits the result to the next processing element. Where a lot of values are calculated, it is not crucial how long and of what topology the pipeline is. More important is the fact that the system produces one output every clock cycle.

Note that pipelining increases the throughput but does not decrease latency.

2.5.2 Data-parallelism

Data-parallelism takes place when several processing elements or nodes do the same work but on different data streams in parallel.

This principle is referred to as Single Instruction Multiple Data streams (SIMD). The complement is called Multiple Instruction Multiple Data streams (MIMD) where several processing elements do different work on different data streams in parallel.

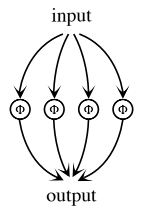

Figure 2.3: Data parallelism

As shown in figure 2.3, the input is split into pieces where every piece is assigned to a different process. The calculation occurs completely in parallel.

2.6 Granularity

Parallel computing distinguishes between between *coarse-grained* and *fine-grained* granularity. While every processing element in a coarse-grained parallelism model calculates relatively big chunks of data, does fine-grained parallelism assign many small, often only single instructions to a processing element. Fine-grained parallelism is used when communication time between processing elements is extremely low compared to the clock frequency.

The functions Φ in the figures 2.2 and 2.3 would in fine-grained parallelism stand for single instructions like an addition or a multiplication whereas they would represent more complex functions in coarse-grained parallelism.

Chapter 3

Mitrion - The virtual processor

The current chapter provides an overview of the Mitrion virtual processor, the programming language Mitrion-C and the reconfigurable hardware on which the Mitrion processor is based on.

3.1 The Mitrion processor

The Mitrion High Performance Computing processor is a *fine-grain, massively parallel* and *reconfigurable* processor that is implemented in FPGAs [9].

The product Mitrion was developed by and is a trademark of the company Mitrionics AB.

Figure 3.1: The Mitrion logo

3.2 Reconfigurable hardware

A clear advantage with reconfigurable hardware is that data dependency graphs can be directly mapped onto hardware. The configuration is specially designed for computing a specific problem. While a sequential processor can only calculate one element in a dependency graph at a time[1], a specially configured processor can operate on several or possibly all nodes simultaneously. Furthermore, the classical von Neumann processor, in contrast to the customized processor, needs to fetch and decode consecutive instructions because of its general purpose. Reconfigurable processors are referred to as FPGAs, which is the topic of the next subsection.

[1]but possibly in a pipelined fashion

3.2.1 FPGA

Field Programmable Gate Arrays (FPGAs) are semiconductor devices with configurable logic components and interconnections. In contrast to *Application Specific Integrated Circuits* (ASICs), FPGAs can be reconfigured after the manufacturing process, which allows them to be customized for specific algorithms and operational areas. This makes them advantageous compared to ASICs in terms of developing costs and time to market.

The major drawback that entails the feature of the reconfigurability is that the devices run at lower clock frequencies than ASICs. It is therefore common to use FPGAs for developing and testing implementation designs for ASICs since ASICS become cheaper as the production volume exceeds approximately 10'000 exemplar [1].

FPGAs have configurable I/O systems that allows them to be deployed in different ways and on different systems. They act as co-processors, where they are used for super-computing. The computationally most intensive parts of procedure are then accelerated and performed by the customized FPGA while the main processor provide the FPGA with data and collect the computed results again. Note that there exist systems with more than one FPGA and such that have more than one CPU per FPGA.

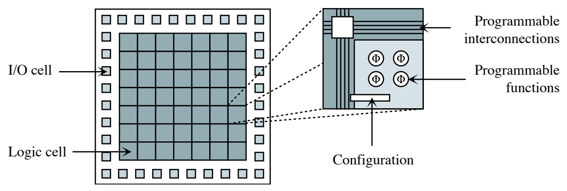

Figure 3.2: General structure of an FPGA by [1]

Vendors

With a market share of 50% is Xilinx [10] the biggest company manufacturing FPGAs [1] today. They recently released their new flagship FPGA *Virtex4*.

3.3 The Mitrion SDK

The Mitrion processor is *reconfigurable* as it is based on FPGAs acting as co-processors. Every configuration is customized for the specific service the system should provide.

It is *massively parallel* because a high number of processing elements on the FPGA can execute in parallel.

The platform is said to be *fine-grained* because every processing element is assigned only a very small part of work, often only one single instruction.

3.3.1 Development process

The Mitrion virtual processor is reconfigured for every service it should provide. The configuration is done by compiling a Mitrion-C program into VHDL code by

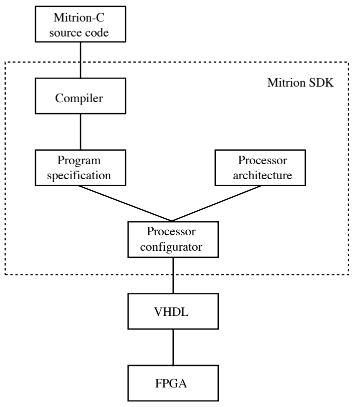

Figure 3.3: Schematic overview of the Mitrion SDK by [2]

the means of the Mitrion SDK. The VHDL code in turn is used to perform the actual mapping of the program onto hardware (place-and-route). This mapping procedure is particular for every FPGA and is done by the tools provided by the vendor of the FPGA.

Mitrion-C programming can be done with little knowledge in hardware. The only constraints, which the programmer should be aware of are the size of the FPGA[2], the internal and external RAM capacities and the architecture of the I/O system[3].

The programmer can optionally use external functions, which are directly written in VHDL.

Once the FPGA based co-processor is configured, it can be used by the host program, which is run on the master processor. The host program is written in ANSI-C and uses the Mitrion host abstraction layer (Mithal) API to interact with the FPGA.

The host program can also use the Mitrion simulator as an FPGA simulator, which allows the entire application to be debugged before a time consuming place-and-route is performed.

3.3.2 Target platforms

Supercomputers that are supported by Mitrion are the *Cray XD1*, various platforms using FPGAs by Xilinx and the Silicon Graphics computers *RASC AFINA* and *RC 100*.

The programs discussed in later chapters are designed and optimized for Xilinx Virtex II based platforms. The Virtex II has a total number of 67584 Flip-Flops, 144 internal RAM banks and four external RAMs, each with a bus-width of 64 bits. In order to be sure a place-and-route can be done successfully, the Flip-Flop usage of a program should, as a rule of thumb, not exceed 55 to 75 % of the available Flip-Flops. The clock frequency is 100 MHz.

[2]number of available Flip-Flops

[3]bit-width to the external RAM banks and the number of external RAM banks

3.4 The programming language Mitrion-C

Today, the most common languages for designing hardware (i.e. configure FP-GAs) are Verilog [11] and VHSIC Hardware Description Language VHDL[12]. Programming sophisticated configurations using these languages is quite time intensive because the languages are very much low-level. Mitrion-C on the other hand is used to write FPGA configurations despite the fact it is not a circuit design specification language in any way. It is much easier to program since it has a higher abstraction level. Although it is called Mitrion-C it is little related to ANSI-C. It is a C-family language but it is about as similar to C as Java is similar to C [13].

Mitrion-C describes data dependencies rather than order-of-execution and allows fine-grain parallelism to be described. The language allows a complete data-dependency graph to be created from the program [13].

The next subsections present the basics of Mitrion-C. For detailed specifications please refer directly to *The Mitrion-C Programming Language* manual [9].

3.4.1 Types and Assignments

Since Mitrion-C is used for hardware programming it consists almost only of a set of primitives.

Scalar types

Scalars may be one of the following: **int** (integer), **uint** (unsigned, or positive integer), **bool** (boolean variable, accepts only the values **true/false**), **float** (floating point variable) or **bits** (a raw binary word of bits).

When declaring a variable of one of the above listed types, the programmer has also to specify the *bit-width* of that variable (except for booleans, which has always a bit-with of 1). Since the surface of an FPGA is quite limited the programmer can specify any number for the bit-with in order to save hardware resources.

Assignment

An example for declaring a variable of type **int** with bit-width *8* named *myint* with the assigned value *3* is stated as follows:

```
int:8 myint = 3;
float:24.8 myfloat = 0.0;
```

The second statement declares a floating point value. For floating point values the *mantissa-width* and the *exponent-width* must be declared. The above example declares an IEEE-754 single precision floating point variable with 24 bits mantissa and 8 bits exponent. To every variable a value is assigned only once and at the time of creation. This principle of single-assignment distinguishes Mitrion-C from other programming languages. The idea behind is that the language is used to implement dependency graphs. See section 3.4.2 for more details.

Collection Types

The two collection types in Mitrion-C are lists and vectors. Collections can be multi-dimensioned where each dimension can be of different size and type[4].

Both data structures can be processed in different ways. Elements in a vector can be accessed by index, which is not possible for lists.

Lists can be reformatted to vectors and vice versa. In addition can multidimensional lists or vectors be reshaped to any desired combination of dimensions, where the total number of elements must of remain the same.

3.4.2 Dependency and functional aspect

In principle, if an instruction is not dependent on previous calculations, it is executed directly. Order-of-execution can be controlled in the way that dependencies among several instructions are explicitly created.

Generally, every instruction or every block of instructions returns a result, which can be of one of the above introduced types and collections. If an expression is dependent on a return variable from a previous block or statement, it can only be executed after the variable has been calculated. In this way can order-of-execution be controlled.

3.4.3 Loops

Mitrion-C provides three types of loops, namely the **while**, the **for** and the **foreach** loops. They are useful for to process collections. Each of them works differently when used to process different collection types.

Foreach loop

Where no dependencies between loop iterations exist, the **foreach** loop is used. When iterating over a vector with the **foreach** loop, all elements are processed in a data-parallel manner (wide parallel). Iterating over a list with a **foreach** loop on the other hand occurs in a pipelined way.

For loop

If every iteration is dependent on the results of its previous, the **for** loop must be used. Iterating over a vector with this kind of loop provokes the vector to be unrolled. Processing a list with the **for** loop occurs in a sequential manner.

While loop

The **while** loop is very similar to the **for** loop except of the property that it iterates only as long as the provided condition is true.

3.4.4 Example: Fibonacci sequence

The Fibonacci sequence is obtained by starting with 1 and 1 and then adding subsequently the two previous Fibonacci numbers to get the next Fibonacci

[4]vector or list

number. The Mitrion-C code that calculates the first 20 Fibonacci numbers looks as follows.

```
Mitrion-C 1.0;
uint:20<20> main()
{
  uint:20 fib = 0;
  uint:20 prev = 1;
  fibonacci = for(i in <1..20>)
  {
    prev = fib;
    fib = fib + prev;
  } >< fib;
} fibonacci;
```

As already mentioned, when declaring a variable, the programmer must specify the bit-width. Because the program does only produce the first 20 Fibonacci numbers, it is known that the last number will not exceed 2^{20}. Hence, the bit-widths 20. The returned result is a list of unsigned integers of the same bit-width. The symbol >< before the returned value means that the whole sequence and not only the last value is returned from the loop.

3.4.5 Memory management

The Mitrion processor can fetch and store data via RAM banks. The number and the volume of internal and external RAM banks varies among different FPGAs and their environment. Mitrion-C uses instance tokens in order to control the memory access succession. Every time a memory is read or written, the call returns an instance token[5]. Similarly is an instance token passed as an argument when accessing a memory. In this way, race conditions can be avoided. A memory write and read sequence could look as follows:

```
token1 = _memwrite(token0, index0, value0);
(value1, token2) = _memread(token1, index1);
```

3.5 Graphical debugger

The graphical debugger is a good aid to notice at once if the program was or not optimally designed. The debugger runs the compiled Mitrion-C program in simulation mode in which the execution can be traced step by step. It also includes a troughput analysis function, which indicates possible bottlenecks and how severe they are. A screenshot is found in the appendix A.

[5]additionally to a value in case of a read

Chapter 4

Matrix multiplication

This chapter concerns a Mitrion-C solution for matrix multiplication. It contains a presentation of the algorithm as well as a performance analysis including a comparison with an ANSI-C implementation run on a general purpose PC.

4.1 Matrix multiplication

Multiplying two matrices is computationally intensive where the matrices are not of trivial dimensions. On the other hand claim applications a lot of computing power where a high number of small matrices is multiplied. This is for example the case in 3d applications, which multiply large numbers of 4×4 matrices for vertice transformation.

Multiplying an $n \times m$ matrix A with an $m \times l$ matrix B yield an $n \times l$ matrix C. The only constraint is that the number of columns of A is equal to the number of rows of B. The equation below exemplifies a multiplication of two 4×4 matrices.

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \times \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix}$$

The mathematical formula for the calculation is given by 4.1.

$$\forall(i,j) \in [0, n-1] : c_{ij} \Leftarrow \sum_{k=0}^{n-1} a_{ik}b_{kj} \tag{4.1}$$

An generic procedure implementing 4.1 is written in algorithm 1.

4.1.1 Complexity

Multiplying two square matrices A and B, each of size $n \times n$, yield a matrix C of the same dimensions. A serial program solving this problem has three nested loops, each iterating n times, thus the complexity is of order $\theta(n^3)$.

Algorithm 1: MATRIX MULTIPLICATION $AB = C$

```
1. for $i := 0$ to $heightA$ do
2.   for $j := 0$ to $widthB$ do
3.     $temp := 0$
4.     for $k := 0$ to $widthA$ do
5.       $temp := temp + A[i,k] \times B[k,j]$
6.     end for
7.     $C[i,j] := temp$
8.   end for
9. end for
```

4.2 Implementation in Mitrion-C

Depending on the size of the matrices, different performance results can be achieved. In case that all multiplications of a row of A with a column of B can be executed in parallel, the order of the problem complexity can be reduced from $\theta(n^3)$ to $\theta(n^2)$ if the elements can be summed in a pipelined fashion after the data-parallel multiplication step. This corresponds to a theoretical speedup of $\theta(n)$.

The major problems for doing so are the limited FPGA resources and the memory bandwidth because there can only be read 4×64 bits per clock cycle, which corresponds to 4 double precision or 8 single precision floating point values. This limitation has a huge impact on the performance since all elements of A and B have to be read several times. Two implementations were made. One with subsequently loading the needed elements from the outer memories and a more interesting and optimized version, which loads the entire matrices A and B into the internal RAM banks prior to execution of the multiplication. The next subsections focus on the latter variant.

4.2.1 Small square matrices

In case of relatively small matrices, a row of A can be multiplied with a column of B completely in parallel. A Virtex II FPGA allows such an implementation for single precision matrices with dimensions up to size 32×32. To evade the memory bandwidth limitation during calculation, the matrices were entirely loaded into the internal memory banks prior to execution of the actual algorithm. A and B are stored in a row-wise and a column-wise fashion, respectively. Like this, every row of A and every column in B can be accessed within one clock cycle.

Since the generic algorithm could be reduced from three to two nested loops there is a high performance gain. Nevertheless, one has to take into account that the discussion up to now was about relatively small matrices. With this design the processor has an output of approximately one entry of the resulting matrix C every clock cycle.

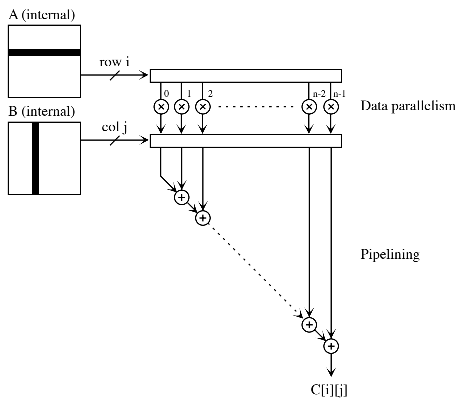

Figure 4.1: Matrix multiplication dependency graph

4.2.2 High dimensioned square matrices

An implementation was made, which is suitable for matrices A and B of such dimensions, that they both fit in the internal RAM banks. The matrices are preloaded into the internal memory banks before the actual calculation begins. For the internal memory consumption it is important what the bit-width per value is. In case of single precision matrices with 32 bits per entry, a matrix of dimensions 128×128, which has 12384 entries in total, needs 524288 bits (65536 bytes or 64KB) of memory. A double precision matrix would have consumed 128KB instead.

The matrices have to be stored in a way that 32 elements from each matrix can be accessed simultaneously. One matrix must be stored row-wise, the other column-wise. Figure 4.2 shows a row-wise storage scheme where 32 elements in a row can be accessed simultaneously (from the internal memory banks). There are 32 different internal memory banks needed.

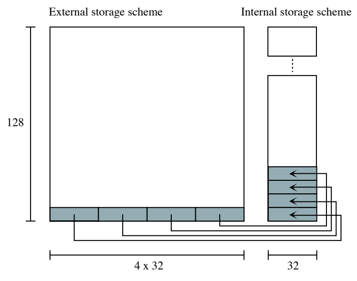

Figure 4.2: Storage scheme for a 128×128 matrix.

The actual calculation is performed in an analogous way as depicted in figure 4.1. The difference is that the output $C[i][j]$ (in figure 4.1) is only a part of the final entry $C[i][j]$. If the matrix dimensions are $n \times n$ there are $n/32$ successive results like in figure 4.1 added for obtaining one entry of the resulting matrix C.

For this implementation, the only constraint for the matrix dimension n is that n is divisible by 32.

4.2.3 Matrix preloading

If the matrices consist of single precision entries, there can be read two values per external RAM and clock cycle. Since the FPGA can read 64 bits per external RAM and clock cycle, the two values are first read as a value pair in raw bits. The 64 bit vector is then split into two 32 bit vectors where each of them is converted to single precision floating point values. With a total number of 4 external RAMs, there can be 8 matrix entries be loaded per clock cycle.

The whole preloading procedure consumes approximately $\frac{n^2}{8}$ clock cycles per matrix. Given a $10ns$ clock (100 MHz), loading a 128×128 matrix takes about $20.48\mu s$.

4.2.4 Exploit the FPGA to full capacity

An implementation, where 32 single precision floating point entries of A and B are multiplied in parallel, needs 41230 flip-flops, which corresponds to a usage of 61% of a Virtex II.

Using double precision floating points instead needs 39598 flip-flops for half the degree of parallelism (16 multiplications in parallel). An option would be to use less costly fixed point numbers. A new type, which Mitrion-C may provide in the future. So far can fixed points be represented by standard integers.

Not only decreases the degree of parallelism with the augmented bit-width, the application will also become slower because of the matrix preloading time, which will be doubled comparing to the single precision version. This is however not the most time consuming part compared to the entire procedure of matrix multiplication.

Once the matrices are stored internally, the estimated execution time for the multiplication algorithm is given by the approximation:

$$\left(\frac{n}{n_{par}}\right) \times n^2 \times clock \tag{4.2}$$

where n is the dimension of the matrices, n_{par} is the number of elements that can be multiplied in parallel and $clock$ is the time per clock cycle. Multiplying two 128×128 matrices where 32 elements can be multiplied in parallel takes by this estimation formula $655\mu s$, assumed a $10ns$ clock.

4.3 Performance

Experimental results yielded the following. Multiplying two 128×128 matrices with single precision floating point numbers uses $682\mu s$ on the Mitrion processor. An equivalent ANSI-C program on the AMD 64 3200+ processor needed $15ms$ for execution.

The execution times for the given size of the matrices A and B are repeated below:

Matrix size	**C**	**Mitrion**	**speed-up**
128×128	$15ms$	$682\mu s$	$21.99\times$

The speed-up of the Mitrion processor over the AMD can only be explained by the constant and simultaneous internal memory access times of the Mitrion processor and the thereby introduced parallelism. The AMD suffers apparently caching problems when reading and writing data from and to the main memory.

The algorithm with preloading the matrices A and B used the following amount of resources:

Precision	**FPGA**	**Flip-Flops**	**RAMs**
Single	Virtex II	41230 (61%)	64 (44%)

61% is a rather high number but is still in the permitted range between 55% and 75%.

4.4 Summary

The benchmark concerned a multiplication of two square matrices of sizes 128×128 using single precision floating point numbers. To attenuate the impact of the memory bandwidth limitation were the matrices entirely loaded into the internal memory banks of the Mitrion processor. Like this it was possible to access and multiply as much as 32 elements of either matrix simultaneously. By this concept could the matrix multiplication be accelerated about $22\times$ comparing to the ANSI-C program run on the sequential processor.

Chapter 5

Gaussian elimination

This chapter presents the Gaussian elimination algorithm, which used to solve systems of linear equations and elucidates the suitability of the algorithm for implementation in Mitrion.

5.1 Introduction to Gaussian elimination

The problem of solving n linear equations with n unknowns is formulated as $Ax = b$ where A is a square matrix of dimension $n * n$ and b is vector with n columns and x is the vector of size n containing the unknown variables. In fact, it is not necessary to have n equations and n unknowns. A solution can exist if the number of equations is higher than or equal to the number of unknowns. However, this chapter considers the matrix A to be square.

One standard procedure to solve $Ax = b$ is to apply Gaussian elimination in a first step as analyzed in this chapter. If $det A \neq 0$ the matrix is not singular and has a unique solution to the problem $Ax = b$ for any vector b.

After Gaussian elimination has been performed successfully, back substitution is used to yield the actual solution. This part however is not subject of this chapter.

5.2 Basic algorithm

The Gaussian elimination algorithm has three nested loops. Several variations of the algorithm exist, depending on the order in which the loops are arranged. A generic procedure implementing Gaussian elimination is found in algorithm 2[1].

5.2.1 Partial pivoting

In case the pivot of the active row, say element $A[k][k]$, is null, the algorithm stops because one can not divide by zero. If this happens, a row interchange must be performed. Note that this does not affect the final solution as row interchangement leaves the system row-equivalent. From all the rows in the

[1]without partial pivoting

Algorithm 2: GAUSSIAN ELIMINATION FOR $Ax = b$

```
**for** $k = 0$ to $n - 1$ **do**
  **for** $j = k + 1$ to $n - 1$ **do**
    $A[k, j] := A[k, j]/A[k, k]$
  **end for**
  $x[k] := b[k]/A[k, k]$
  $A[k, k] := 1$
  **for** $i = k + 1$ to $n - 1$ **do**
    $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$
  **end for**
  $b[i] := b[i] - A[i, k] \times x[k]$
  $A[i, k] := 0$
**end for**
```

active part a row is selected to be interchanged with the failing pivot row. This row should be the one with the highest absolute value of the elements in column k below row k.

5.2.2 Complexity

Since three nested loops are needed, each iterating at maximum n times, the complexity for the serial algorithm is of order $O(n^3)$, where n is the dimension of the matrix A.

5.3 Parallelization

In the generic algorithm there are a number of instructions, which can be executed independently in parallel. First of all is the division step, also referred to as *normalization*. Dividing all elements in the current row (pivot row) by the first one (pivot), can occur concurrently because there are no dependencies among the divisions.

Second, after the normalization of the pivot row has been done, all the elimination steps of the rows below can be carried out completely in parallel.

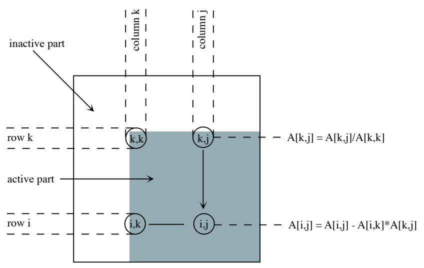

Figure 5.1: Gaussian elimination

5.4 Implementation on the Mitrion platform

On a first glance one could think an FPGA being an excellent solution for performing Gaussian elimination because of its fine-grained granularity. The division step in the code above can theoretically divide all elements in a row in parallel. The following elimination step can be carried out concurrently as well, where every element of the respective row can be updated at the same time.

With this assumptions, one iteration of the outer loop would take constant time $O(1)$! Having a total of n outer loop iterations, the Gaussian elimination would be reduced from $O(n^3)$ to $O(n)$! This corresponds to a speedup of $O(n^2)$!

The reason why this is only theoretically possible is because an FPGA has a limited number of processing units. Hence, it is not possible to treat an entire matrix of non trivial size on one chip at the same time. The elements have therefore to be successively loaded from the memory, updated and written back to the memory again. This introduces some additional operations, it can however be performed in a pipelined fashion to hide latency.

The main reason why the algorithm in its generic form is not feasible to implement in Mitrion-C is because of the fact that the language does not permit variable loop lengths. In every iteration, the active part of the matrix gets smaller as depicted in figure 5.1. Loops of the following form are not supported in Mitrion-C. Something which may be added in the future.

```
for k=0 to n do
  for i=k to n do
    ...
  end for
end for
```

Because of dependencies among elements, the algorithm does not allow the matrix A to be divided into smaller sub-matrices, where each sub-matrix could be calculated separately. One problem that intervenes here is the partial pivoting.

In literature, algorithms for parallelizing Gaussian elimination are mostly designed for coarse-grained parallel computing environment, something which is not convenient for a single FPGA. The author did not find any solution that could avoid the variable loop lengths. However some alternative algorithms to Gaussian elimination were explored as mentioned in the next section.

5.5 Alternative solutions

Basically there exist two categories of algorithms that can solve the problem $Ax = b$; *direct methods* and *iterative methods.* Gaussian elimination is a direct method. A closely related solution is the LU factorization. The most common iterative solution is the one proposed by Jacobi.

5.5.1 LU factorization

An alternative to the Gaussian elimination is LU-factorization where A is decomposed in a a lower triangular matrix L and an upper triangular matrix U such that $A = LU$. The decomposition makes it possible to calculate x in two steps, namely $Ly = b$ and $Ux = y$. Because L and U are triangular matrices, solving the two equations results in doing back substitution twice.

There exist variations of decomposing A into L and U. But all procedures found by the author were not possible to implement in Mitrion-C because of the same dynamic problem that averted implementation of Gaussian elimination.

5.5.2 Jacobi's iterative method

Solving a system of linear equations can not only be done by direct methods such as Gaussian elimination. There exist algorithms that propose iterative solutions, which turned out to be more adequate for FPGAs. Chapter 6 devotes the iterative method proposed by Jacobi.

5.6 Result

As already mentioned, no variant of the generic algorithm that the author has seen seemed to be mated for Mitrion-C. Quite a lot of time was invested in finding a variation of the original algorithm to go round the dynamic of the algorithm. But after a number of attempts, the author did not do any further experiments because of the above explanations. Herewith is the discussion about Gaussian elimination completed.

Chapter 6

Jacobi's linear equation solver

This chapter describes the Mitrion-C implementation of the Jacobi's iterative linear equation solver. Theory, performance analysis and a comparison with an ordinary PC solution are embraced.

6.1 Motivation for Jacobi's linear equation solver

The problem of solving n linear equations with n unknowns is formulated as $A\mathbf{x} = \mathbf{b}$ where A is a matrix of dimension $n \times n$ and $\mathbf{b}$ is vector with n columns and $\mathbf{x}$ is the vector of size n containing the unknown variables. As discussed in chapter5, are direct methods for solving this problem not suited for Mitrion-C. The iterative method proposed by Jacobi on the other hand is expected to perform well on the Mitrion processor.

6.2 Overview of the Jacobi method

Since the mathematician is interested in the unknowns x_i of the vector $\mathbf{x}$, we use $\mathbf{x}$ initialized with zeros as input to a function of the form $\mathbf{x}^{(\delta+1)} \Leftarrow f(\mathbf{x}^{(\delta)})$ which produces an estimate of $\mathbf{x}$. The estimate $\mathbf{x}$ is the new input of the function in the next iteration. This procedure is repeated until the vector $\mathbf{x}$ converges.

Let $A = L + U + D$, where L and U are the lower triangular and upper triangular matrices of A, and D is the matrix containing only the diagonal entries of A. The actual calculations are as follows:

$$\mathbf{x}^{(\delta+1)} \Leftarrow D^{-1}[\mathbf{b} - (L+U)\mathbf{x}^{(\delta)}] \tag{6.1}$$

or for one element x_i the equation is

$$x_i^{(\delta+1)} \Leftarrow \frac{1}{a_{ii}}[b_i - \sum_{j=1:j\neq i}^{j=n} a_{ij}x_j^{(\delta)}] \tag{6.2}$$

A generic algorithm looks as follows:

Algorithm 3: JACOBI'S LINEAR EQUATION SOLVER $A\mathbf{x} = \mathbf{b}$

```
Require: $A$ is square and strictly diagonally dominant
1.  for $iterations = 0$ to MAX do
2.    for $i = 0$ to $n$ do
3.      $sum = 0$
4.      for $j = 0$ to $n$ where $i \neq j$ do
5.        $sum := sum + x[j] \times A[i][j]$
6.      end for
7.      $x_{iterated}[i] := (b[i] - sum)/A[i][i]$
8.    end for
9.    $x := x_{iterated}$
10. end for
```

6.2.1 Convergence criteria

Because not every system of linear equations converges to the desired vector of unknowns $\mathbf{x}$ there is need for a convergence criteria that ensures the method finds the solution. The method proposed by Jacobi converges only in case the following formula holds for all rows of the matrix A.

$$|a_{ii}| > \sum_{i \neq i} |a_{ij}| \tag{6.3}$$

A matrix holding the criteria 6.3 is said to be strictly diagonally dominant.

6.2.2 Complexity

In every iteration there are two nested loops of length n, which corresponds to complexity $\theta(n^2)$ per iteration.

6.3 Implementation design

The implementation described solves a system of 64 linear equations with 64 unknowns. However the program can be adapted to other problem sizes, as long as the matrix A fits entirely in the internal memory banks.

The calculations are data intensive because every entry of the entire matrix A has to be read once per iteration. As the bandwidth to the external memories is very limited, the entire matrix A was preloaded into the internal memory banks. In case of a 64×64 matrix with double precision values, only 32 KB are needed for storage.

Analogously to the proposed solution in chapter 4, is the matrix A preloaded and stored as depicted in figure 6.1. In order to be able to access a chunk of 8 elements at once, all 8 elements in a row are stored in different memory banks that can be accessed simultaneously.

The unknowns x_i are stored the same way. There is no need to preload the values of $\mathbf{b}$, they can remain in the external memory since they do not constitute a bottleneck.

The preloading time for A is estimated to be $\frac{n^2}{8}$ clock cycles, where $n \times n$ is the dimension of A.

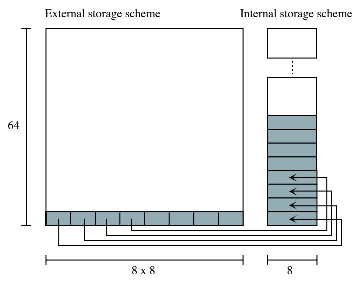

Figure 6.1: Storage scheme of a $64 * 64$ matrix A.

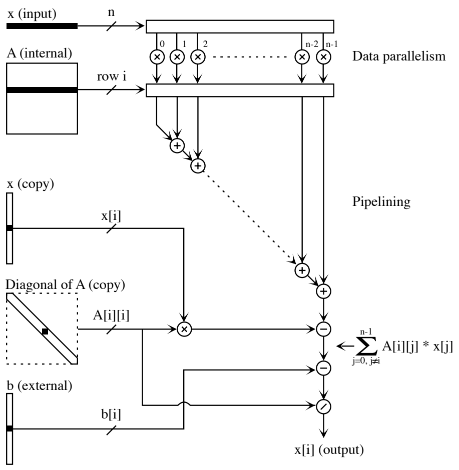

Figure 6.2: Implementation scheme and data dependency graph

Figure 6.2 depicts the dependency graph for only one iteration. The result is stored in a buffer and written back to the memory after the iteration has completed. This is however not shown in figure 6.2.

Assuming the data being preloaded, the time used for one iteration will approximately be (ignoring the pipelining latency):

$$\left(\frac{n}{n_{par}}\right) \times n \times clock \tag{6.4}$$

In case of a 64×64 matrix where 8 elements are multiplied concurrently, the roughly estimated execution time for one iteration is $\frac{64}{8} \times 8 \times 10ns = 6.4\mu s$, assumed a $10ns$ clock.

6.4 Jacobi iterations on a single processor

Something which Mitrion-C does not permit is to read in every iteration data from a different memory location. This is why there is need to store the new vector **x** in a buffer and write it back to the memory after every iteration.

Writing back the buffer can only occur after the last value has been calculated. This provokes the whole pipelining system to be interrupted, i.e. to be emptied and filled up again in the next iteration.

This is can easily be avoided in an ANSI-C program by reading from one memory in an even iteration and from the other in an odd iteration. Analogous can the iterated vector be stored in the opposite location, of course. However only a negligible amount of time can be saved compared to the entire workload of the ANSI-C program. The impact of this lacking feature in Mitrion-C is much bigger for the Mitrion processor.

6.5 Performance

The table below shows the experimental execution times for the calculation of a system with 64 equations with 64 unknowns (64×64 matrix) with double precision where 8 elements are multiplied concurrently.

Iterations	**AMD**	**Mitrion**	**speed-up**
1000	$48ms$	$7.6ms$	$6.32\times$

The resource usage for this implementation on the Virtex II FPGA looks as follows:

Precision	**FPGA**	**Flip-Flops**	**RAMs**
Double	Virtex II	39000 (57%)	50 (34%)

6.6 Related projects

An equivalent iterative Jacobi solver was implemented in VHDL by Morris and Prasanna [14] on an FPGA using identical IEEE 64-bit floating point values. Their design is very similar to the one presented here.

One iteration with double precision and a matrix of size 64×64 has an estimated execution time of $9\mu s$ to load the matrix onto the FPGA and $8.6\mu s$ for one iteration. One has to take into account that they deployed a Virtex II pro FPGA with a $13ns$ clock, which augments the execution time by a factor of 1.3 compared to the Mitrion processor. However the execution times were theoretically established. Nothing was stated about experimental run time.

They implemented a reduction tree to sum all the elements of the multiplication step. This is an optimization, which reduces the latency of calculating the sum of the multiplications from n to $\log(n)$. However, since the number of elements to sum after a multiplication step is only 8, their design saves only a few clock cycles.

Something, which is not included in their calculations is the handling of the iterated vector $\mathbf{x}$. The Mitrion-C program buffers the iterated values x_i and writes them back to the memory after the last value was calculated. This procedure adds at least another 64 cycles to every iteration. Adding 64 cycles to the execution time of Morris and Prasanna's Jacobi solver lets the execution time of their implementation for one iteration become $9.4\mu s$. Considering the lower clock frequency, the execution time would be converted to $9.4\mu s/1.3 = 7.21\mu s$ for a system with a 100MHz clock.

The Mitrion processor calculates 1000 iterations for the same problem size in $7.6ms$, which corresponds to $7.6\mu s$ per iteration. This execution time however includes the matrix preloading and the writing back of the final result to external memory. So as a quantitative evaluation, one can say the implementations being equivalent.

6.7 Conclusion

As seen in the previous section, the Mitrion-C program can not make it faster than an implementation in VHDL. However, it is neither slower. The author suspects much more to have implemented the Mitrion-C version in significantly less time than what would have been needed for a VHDL implementation.

6.8 Summary

The discussed Mitrion-C Jacobi linear equation solver could iterate about 6 times faster than the equivalent ANSI-C program run on the AMD 64 3200+ processor. The speed-up was achieved because the Mitrion-C program could multiply 8 64-bit floating point values in parallel due to the manner how the matrix A was stored in the internal memories. In addition could all summing occur in a fully pipelined fashion. Furthermore was the Mitrion processor as fast as an implementation made directly in VHDL.

Chapter 7

Discrete wavelet transformation for image compression

This chapter deals with the theory, implementation and evaluation of the discrete Haar-wavelet transformation on the FPGA-based platform Mitrion. The performance analysis includes a quantitative comparison with an equivalent ANSI-C program run on an ordinary PC.

7.1 Wavelet transformations

Wavelet transformations are mostly used in signal processing. An image can be seen as a special kind of a signal where the temporal component is replaced by a two dimensional and discrete spatial component.

The wavelet transformation of an image does itself not diminish the amount of data. It does rather transform the image to a set of data, which can be encoded much more efficient than the original image.

An image can be transformed in different ways. Characteristics for wavelet transformations are the number of transformation levels, the order of transformation, the loss of information and the kind of filters used. The filters in turn are derived of the actual so-called mother-wavelet.

Figure 7.1 illustrates the general schema of the image wavelet transformation using high-pass (HP) and low-pass (LP) filter-banks.

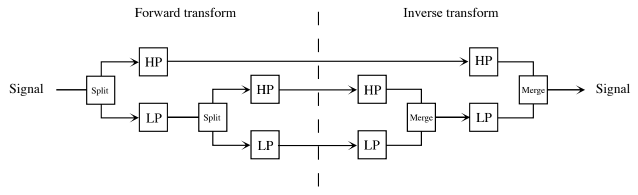

Figure 7.1: Filter-banks in the two-level wavelet transform

A transformation level filters the input signal with a HP and a LP filter. This turns the input signal into approximation (output of LP) and detail (output of HP) coefficients.

The LP filtered part of the signal can be seen as a coarser representation of the original signal, i.e. a representation with lower resolution. The HP filtered part of the signal in contrast constitute the detail information of the original signal. The aim of the transformation is to obtain detail coefficients with very low entropy because they can then be encoded and compressed with a high compression rate.

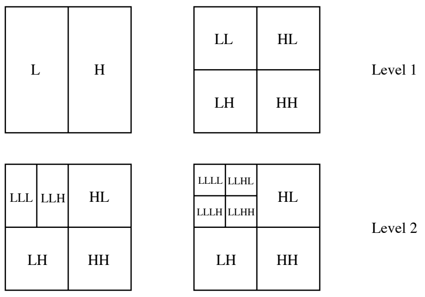

Figure 7.2: Order of wavelet transform

Figure 7.2 shows the transformation order of level one and level two of a two dimensional signal (image). Depending on the kind of wavelet transformation (mother-wavelet), the filtering is done in different ways. Every HP and LP filter pair has different properties regarding entropy reduction. The simplest and most parallelizable filter pair is derived from the Haar wavelet, hence the name Haar wavelet transformation. The next section introduces the concept behind this kind of transformation.

7.2 The Haar wavelet transformation

This section adopts the explanations of Schroeder and Sweldens's presentation [15].

Two neighboring numbers a and b in a sequence can be restated in terms of their average s and difference d.

$$s = \frac{a+b}{2} \tag{7.1}$$
$$d = b - a \tag{7.2}$$

The original numbers a and b can be recovered by:

$$a = s - d/2 \tag{7.3}$$
$$b = s + d/2 \tag{7.4}$$

This observation is the key behind the so-called *Haar* wavelet transformation.

A discrete input signal of length 2^n is by this technique split into two signals s_{n-1} and d_{n-1}, each of length 2^{n-1}. Given the averages and differences, the original sequence can always be reconstructed.

One can think of the averages s as a lower resolution of the input signal and the differences d as the high-resolution coefficients when reconstructing the original sequence from the low resolution sequence.

The same transformation can be applied to the coarser sequence s_{n-1} once again. The result is an even coarser representation of the original sequence and another difference signal, both of length 2^{n-2}. This procedure can be repeated n times until one runs out of samples.

7.3 Thresholding and lossy transformation

In addition to the LP filtering can the obtained coefficients be set to zero if they do not exceed a certain threshold. The coefficients, which are lower than the chosen threshold can be seen as not significant detail information. Setting the concerned coefficients to zero increases the achievable compressing rate (decreases the entropy) but makes a prefect reconstruction of the original image impossible. The choice of the threshold is a trade off between compression rate and quality of recovery of the original image.

Since the wavelet transformation itself is in focus of this discussion, the thresholding was not implemented. It would however be easy to add, and would increase the latency of the program by only a few clock cycles.

7.4 Implementation design

The implementation made in Mitrion-C performs a two dimensional and two level discrete Haar wavelet transformation. The input are 8 bit unsigned integers, which represent gray scale pixels of the image of size 1024×768.

The pixels of the whole image are externally stored in a one dimensional array. The image is like this only read from one external memory bank. The platform used here was an SGI rasc 6000 that has two external memory banks, each 128 bits wide.

Because one memory bank provides the processor with 128 bits per clock cycle, there can $128/8 = 16$ pixels be read per clock cycle.

7.4.1 Integer to integer transformation

In order the LP and HP filtered coefficients to be 8-bit integers again, the transformation was implemented using unsigned integers. The Haar wavelet transformation can be implemented very easily using only unsigned integers because of the following relations [16].

$$s = \lfloor (a+b)/2 \rfloor \tag{7.5}$$
$$d = b - a \tag{7.6}$$

And the backward transform:

$$a = s - \lfloor d/2 \rfloor \tag{7.7}$$
$$b = s + \lfloor (d+1)/2 \rfloor \tag{7.8}$$

Because of the above relations, the rounding does not cause information to be lost. Nevertheless, if b is smaller than a, the d component will become less than zero, which in turn will be interpreted as a pixel with very high intensity. If for example $b = 0$ and $a = 1$, $c = 0 - 1 \rightarrow 255$, which is represented as a white pixel. The HP coefficients will therefore be mostly black (close to 0) or white (close to 255) pixels. In order to turn the white pixels into black ones, the coefficients could be thresholded in an appropriate way.

7.4.2 Data flow

The input data is an 8-bit gray scale image of size 1024×768. Since one external memory bank can provide 128 bit of data every clock cycle and one pixel is 8 bit wide, there can 16 pixels in a row be read per clock cycle and memory bank. Because it is assumed that the image is placed entirely in one external RAM, only one RAM bank could be used to read the image. The transformed image is written to the other RAM bank to avoid contention.

The reading progression occurs according to the scheme depicted in figure 7.3. The first four chunks are marked with darker grey. They are read in the order as indicated by the arrow. The next 3×4 chunks are marked with lighter grey. The reading is repeated in this manner (4×4 chunks) until the whole image was read.

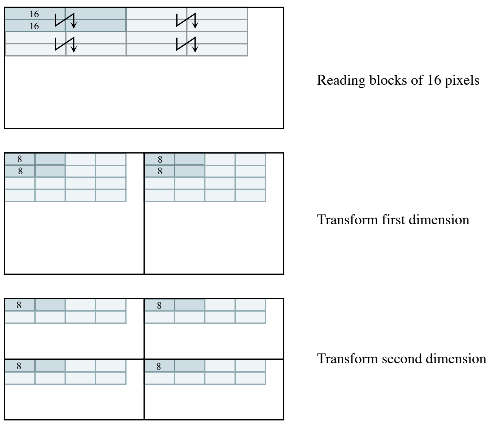
16
16
Reading blocks of 16 pixels
8
8
8
8
Transform first dimension
8
8
8
8
Transform second dimension

Figure 7.3: First level of two dimensional Haar wavelet transform

As can be seen in the last picture of figure 7.3, the HH, HL, LH and LL coefficients contain all 2×8 coefficients in a row, which is a total of 128 bits in each window. As soon as 16 coefficients in a row are calculated, they are converted to a raw 128 bit array and written back to the memory. The order of input reading was chosen to induce this effect after shortest possible execution time.

The exception however are the LL coefficients, which are transformed once again. Since it takes more time to obtain the final coefficients of the LL window (LLLL, LLLH, LLHH, LLHL) and because the whole transformed image is written back to one single external memory bank, it was not possible to synchronize the writing back of the second pass coefficients with the writing back of the LH, HL and HH coefficients. The second pass coefficients were therefore buffered in another memory location and only written back to the target memory after all calculations have been completed.

7.5 Resource usage

The implementation proposed claims not a lot of resources of the FPGA. As much as 24971 flip-flops (36%) and 92 Block Rams (63 %) were needed for the configuration.

One could possibly fit an embedded zero-tree encoding (see [17]) or Huffman coding (see [17]) algorithm on the same FPGA. Adding such an algorithm would implement a complete image compression algorithm. A diploma thesis that concerns a complete coding and decoding algorithm was for example written by D. Bachofen [18].

Platform	**Flip-Flops**	**RAMs**
SGI RASC v6000	23315 (34%)	124 (86%)

7.6 Performance

The complete two-level discrete Haar wavelet transformation used on the Mitrion processor $0.614ms$ for processing an image of size 1024×768. This corresponds to reading 8 pixels per clock cycle plus the latency of the algorithm. With a total of 768×1024 pixels, there are $(768 \times 1024)/16 = 49152$ clock cycles needed to read the whole image. Having a $10ns$ clock (100MHz) the reading corresponds to approximately $0.492ms$. Additional clock cycles are needed to write back the buffer of the second passed LL coefficients. The size of the buffer is 512×384 pixels. Packing 16 pixels into one 128 bit array diminishes the time of transferring the buffer by a factor of 8. Hence $(512 \times 384)/16 = 12288$ clock cycles, which corresponds to $0.12ms$.

Adding up the reading of the image plus the writing back of the buffer gives an estimated execution time of $0.12ms + 0.492ms = 0.612ms$, without considering the latency of the algorithm. This number matches the experimental results.

An equivalent ANSI-C program run on the AMD 3200+ PC used $14ms$ for transforming an image of the same size. The Mitrion processor attained in this case a speedup of $14ms/0.614ms \approx 22.8$.

Image size	**Transformation levels**	**AMD**	**Mitrion**	**speed-up**
1024×768	2	$14ms$	$0.614ms$	$22.8\times$

7.7 Summary

Comparing to the ANSI-C two-dimensional and two-level discrete Haar wavelet transformation on the AMD processor is the Mitrion processor faster by a factor of more than 22 times in transforming an 8-bit gray-scale image of size 1024×768. An important factor, which affected this result was that the Mitrion processor can calculate with 8-bit unsigned integers instead of using standard single precision values. This advantage lessened the impact of the limited bandwidth of the

Mitrion processor in that it could read and write chunks of pixels simultaneously out of and to external memory banks.

Chapter 8

CORDIC

This chapter deals with the theory, implementation and evaluation of the CORDIC algorithm, which is used to calculate the trigonometric functions sine and cosine. The performance analysis includes a quantitative comparison with an equivalent ANSI-C program run on an ordinary PC.

8.1 Overview of CORDIC

CORDIC (COordinate Rotation DIgital Calculation) calculates the trigonometric functions sine and cosine, magnitude and phase (arc-tangent) to any desired precision [19]. The algorithm is especially designed for digital computation.

Only sine and cosine calculation for any angle $\alpha \in [0, 360]$ is covered in this chapter.

8.2 Sine and cosine calculation

Recall from trigonometry that if given a circle with radius one, there can directly be obtained the sine and cosine values of an arbitrary angle α when the point on the circle at α degrees is projected onto the y-axis or x-axis respectively.

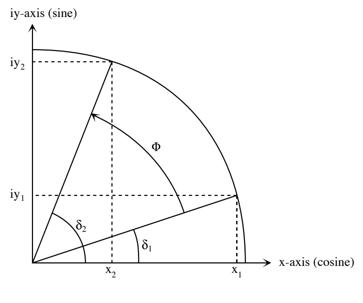

Figure 8.1: CORDIC rotation scheme for sine and cosine calculation

In figure 8.1 x_1 and iy_1 correspond to $\cos(\delta_1)$ and $\sin(\delta_1)$. The same is true for x_2, iy_2 and δ_2.

Since,

$$\delta_2 = \delta_1 + \phi$$

the following identities for the corresponding sine and cosine can be stated:

$$\sin(\delta_2) = \sin(\delta_1 + \phi) = \sin(\delta_1) \times \cos(\phi) + \cos(\delta_1) \times \sin(\phi) \tag{8.1}$$

$$\cos(\delta_2) = \cos(\delta_1 + \phi) = \cos(\delta_1) \times \cos(\phi) - \sin(\delta_1) \times \sin(\phi) \tag{8.2}$$

By replacing sines and cosines of δ_1 by the complex notation iy_1 and x_1 in the previous equations, we the following relation is obtained:

$$x_2 = x_1 \times \cos(\phi) - iy_1 \times \sin(\phi) \tag{8.3}$$

$$iy_2 = x_1 \times \sin(\phi) + iy_1 \times \cos(\phi) \tag{8.4}$$

If the angle δ_1 in figure 8.1 is rotated by ϕ to δ_2 the new and old values for x and iy are related as stated in the equations above.

Because of the property

$$\tan(\phi) = \frac{\sin(\phi)}{\cos(\phi)}$$

the sines in the equations above can be replaced by $\tan(\phi) \times \cos(\phi)$. The resulting equations are the following:

$$x_2 = \cos(\phi) \times [x_1 - iy_1 \times \tan(\phi)] \tag{8.5}$$

$$iy_2 = \cos(\phi) \times [x_1 \times \tan(\phi) + iy_1] \tag{8.6}$$

So far nothing became really simpler. But the last two equations are a good starting point for the numerical sine and cosine calculation.

Starting with any arbitrary angle (δ_1 in 8.1) who's sine and cosine values are known, the CORDIC algorithm rotates the current angle (δ_1 in 8.1) towards the target angle (δ_2 in 8.1) step by step. In every step the sine and cosine values of the current angle are updated by the equations above. This can be done in an efficient way, if $\tan(\phi_i)$ of the rotation angles ϕ_i are fractional powers of 2. In this case, the multiplications can be replaced by simple shift right operations.

Another interesting fact is that $cos(\phi) = cos(-\phi)$. Hence, for the $\cos(\phi)$ factors, it does not matter whether the rotation is clock wise or counter clock wise. Because $\tan(\phi)$ was chosen to be a fractional power of two, the cosine factors become

$$\cos(\tan^{-1}(2^{-i})) = 1/\sqrt{1 + 2^{-2i}} \text{ for iteration } i.$$

If this formula is accumulated over n iterations, one obtains

$$c_n = \prod_{i=0}^{n-1} 1/\sqrt{1 + 2^{-2i}}$$

which is a constant value. The $\cos(\phi)$ factors can therefore be replaced by this constant value, called *aggregate constant*, because the total number of iterations is known and equal to n. Thus, the formulas for a total of n iterations look as simple as

$$x_{i+1} = x_i \mp iy_i \times \tan(\phi_i) \tag{8.7}$$

$$iy_{i+1} = iy_i \pm x_i \times \tan(\phi_i) \tag{8.8}$$

and for obtaining the final result:

$$x_{final} = x_n \times c_n \approx \cos(\alpha) \tag{8.9}$$

$$iy_{final} = iy_n \times c_n \approx \sin(\alpha) \tag{8.10}$$

8.2.1 Precision of CORDIC

The table below lists the first set of values of ϕ such that $\tan(\phi)$ are fractional powers of two.

$\tan(\phi)$	$\phi°$	$\cos(\phi)$
$1/2^0$	45°	0.707107
$1/2^1$	26.5650°	0.894427
$1/2^2$	14.0362°	0.970142
$1/2^3$	7.12502°	0.992278
$1/2^4$	3.57633°	0.998053
$1/2^5$	1.78991°	0.999512
$1/2^7$	0.895174°	0.999869
$1/2^{10}$	0.055952892°	0.999999523
$1/2^{15}$	0.001748528°	0.999999999534
$1/2^{19}$	0.000109283°	≈ 1

The results of every iteration are the sine and cosine values for an angle which is an approximation of the desired target angle. In each iteration, the approximating angle gets closer to the target angle, where the difference between the angles gets almost diminished by a factor of two.

If the starting angle is 0 and the target angle is 90 degrees, the approximation gets as close as 90.0000246 degrees after 20 iterations. This corresponds to a precision of 2.729×10^{-5}.

8.3 Implementation design

This section describes how the CORDIC algorithm was implemented in Mitrion-C.

8.3.1 Parallelism

The CORDIC algorithm of sine and cosine calculation proceeds in an iterative way. Because every iteration depends on the results of its previous, there is unfortunately no possibility to parallelize one single calculation. Hence, one has to calculate a number of values simultaneously in order to introduce parallelism.

8.3.2 Pipelining versus data-parallelism

In case of multiple sine and cosine calculations, the best way is to create a pipeline in which every stage performs one iteration and refines the approximation of sine and cosine. Because every stage in the pipeline updates only one value, the pipeline is fed with one target angle per clock cycle and puts out one sine and one cosine value per clock cycle on the other end. Like this, there is no challenge regarding limited memory bandwidth, assuming that there are at least three external memories available, one for to read the target values and two for to write the corresponding sign and cosine values.

The length of the pipeline affects the accuracy of the output values. A balance has to be found between the bit-width of the intermediate and final results and the number of pipeline stages because a small number of stages permits a higher bit-width and vice versa.

Algorithm 4: CORDIC SINE AND COSINE CALCULATION OF α

```
Require: $0 \leq \alpha \leq 360$
1. $const := 0.607253$
2. if $\alpha > 180$ then
3.   $\phi := 270$ /* starting angle */
4.   $x := 0$ /* $\cos(270) \times const$ */
5.   $iy := -const$ /* $sin(270) \times const$ */
6. else
7.   $\phi := 90$ /* starting angle */
8.   $x := 0$ /* $cos(90) \times const$ */
9.   $iy := const$ /* $sin(90) \times const$ */
10. end if
11. for $i = 0$ to MAX do
12.   if $\phi < \alpha$ then
13.     $\phi := \phi + \tan^{-1}(2^i)$ /* rotate clock wise */
14.     $x := x - iy \times 1/2^i$
15.     $iy := x \times 1/2^i + iy$
16.   else
17.     $\phi := \phi - \tan^{-1}(2^i)$ /* rotate counter clock wise */
18.     $x := x + iy \times 1/2^i$
19.     $iy := iy - x \times 1/2^i$
20.   end if
21. end for
```

Calculating with 64 bit values allows a 12 stage pipeline to be implemented on one FPGA. This is somehow a disaccord because 64 bit values have much higher precision than what can exploited with a 12 stage pipeline. Thus the algorithm was implemented with single precision allowing a pipeline of more than 20 stages. Note that the length of the pipeline is not crucial for the execution time in case a large number number of sines and cosines are calculated.

Where not much accuracy is needed, many shorter pipelines running in parallel can be implemented instead of only one. Of course, this would permit a higher throughput. Today's FPGAs allow this application to be implemented with two data-parallel 20 stage pipelines, each with single precision.

Since Mitrion-C does not have fixed-point types, some kind of pseudo fixed point numbers were used. This was done by simply multiplying all input values by a factor of 10^{10} and converting them to integer values. After execution, the results are converted back to floating point values. Like this, the divisions and multiplications can be replaced by arithmetic shift operations.

As a coarse approximation of the execution time, one could disregard the latency of the pipeline and only consider the time needed for reading of the target angles. There can two values be read per clock cycle. The estimated execution time for the calculation of n target angles will be:

$$\frac{n}{2} \times clock$$

where *clock* is the time per clock cycle. Assuming a $10ns$ clock, the lower bounded execution time for the calculation of 2000 angles will be $10\mu s$ (neglecting the latency).

8.3.3 Unsolved problems

If one wants to calculate sine and cosine of 45 degrees, there would only one iteration be needed[1] because the approximating angle is rotated precisely onto the target angle, since $\tan(45) = 1$, which is a fractional power of 2.

In the next iteration, the approximating angle is rotated away and causes the result to become worse! In every succeeding iteration, this error is diminished again. A countermeasure to address this undesired effect would be to insert a relative costly if-statement in every stage of the pipeline, which checks for this special case. An implementation with if-statements needs more space on the FPGA per pipeline stage, something which causes the pipeline do become shorter. The shorter pipeline in turn diminishes the accuracy of the results in general.

The decision was made not to implement the additional if-statement for this special case where $\tan(\phi)$ of the target angle ϕ is a fractional power of 2 because the precision of the result without special care is equal for any arbitrary target angle ϕ.

8.4 Performance

The table below shows the performance results of the above described implementation. A total of 27202 flip-flops are needed for two 20-stage pipelines running in parallel whilst an implementation with a single pipeline of the same length would have needed only 13836 flip-flops. Having one single pipeline would obviously decrease the speed of the application by a factor of two.

Execution time per	**AMD**	**Mitrion**	**speed-up**
2000 angles	$392\mu s$	$11.19\mu s$	35.63×
1 angle	$196ns$	$5.5ns$	35.63×

The obtained results let the Mitrion processor appear in bright light. At 100MHz it is more than 35 times faster than an equivalent ANSI-C program run on the AMD 64 3200+ processor.

The resource usage for this algorithm looks as follows:

Precision	**Pipelines**	**Flip-Flops**	**RAMs**
Single	2	25818 (38%)	14 (9%)

8.5 Summary

The data-parallel and fully pipelined CORDIC implementation in Mitrion-C beats the ANSI-C program run on the sequential processor by an amazing factor of more than 35 times. The reason for this result is the extremely high degree of parallelism of the Mitrion-C implementation.

[1]assuming to start at 0 or 90 degrees

Chapter 9

Discussion

This chapter reviews pros and cons that were encountered while working with the Mitrion SDK.

9.1 FPGA based platforms

Computer scientists tend to argue about what programming language and what kind of workstation could solve best a certain computationally intensive problem. Oftentimes, one does not consider that the hardware itself can perform various tasks. It turns out that hardware implementations can be even much more efficient in some cases. The performance results of the previously discussed algorithms approve this statement.

In terms of large scale systems, one could name the current project of Mitrionics, which tries to beat a 700 workstation cluster with a 14 FPGA system regarding the genetic BLAST algorithms.

The fact that the speed of FPGAs grows faster than the speed of traditional CPUs let the author expect FPGA based systems to become even more advantageous in the future. In addition grows the size of FPGAs continuously, which makes it possible to implement increasingly complex algorithms on a single chip.

9.2 Hardware design process

So far, hardware programming was to be done using quite low-level programming languages like VHDL or Verilog. Programming in those languages is very time consuming, even for small applications. As problems get bigger, coding VHDL gets very complicated. A simple example is a program that adds two vectors. While about 400 lines of VHDL implement this procedure is the same algorithm programmed in 7 lines Mitrion-C code [3].

Attempts were made to use ANSI-C like languages like Handel-C to describe algorithms in hardware. However, none of the proposed solutions is very popular these days. Mitrionics saw the necessity of designing a completely new language, which concentrates on data dependencies and parallelism. Programming Mitrion-C becomes quite intuitive, once the programmer has acquired the basic knowledge. The complexity of the problems are much easier manageable comparing to VHDL programming. So can for example a 180 lines Mitrion-C program easily generate thousands of lines of VHDL code. Complexity management may become an even more important factor with the growing size of FPGAs. Despite the higher abstraction level of Mitrion-C, the programs are as fast as implementations made directly in VHDL. Furthermore is debugging in Mitrion made easy with the graphical debugger and the simulation mode of the Mitrion processor.

9.3 Controlling resource usage

Another valuable feature of the Mitiron SDK is that the programmer sees immediately an estimation of hardware resources used for the compiled program. The degree of parallelism, like in the discussed Jacobi iterative linear equations solver or the matrix multiplication as well as the length of the pipelines in CORDIC could like this easily be adapted to the available resources. If for example the data-parallel multiplication step of a row of A with a column of B in a matrix multiplication claims too many flip-flops, the **foreach** loop can easily be broken into a **foreach** loop of half the size, which is then executed twice instead of only once.

9.4 Dynamic of Mitrion-C

In spite of all the mentioned advantages of Mitrion, one has also to mention the previously presented Gaussian elimination algorithm, which failed to be implemented in Mitrion-C because of the lack of dynamic support, like illustrated in the code fragment below. The missing feature of dynamic inner loop-length averted the implementation of Gaussian elimination in Mitrion-C.

```
for k=0 to n do
  for i=k to n do
    ...
  end for
end for
```

Problems with high dynamic are not well or not at all suited for Mitrion-C. When implementing recursive algorithms like for example recursive tree evaluation, the programs may only evaluate to a pre-defined level since all function calls are made in-line at compile time.

To attenuate the problem of lack of dynamic in Mitrion-C, one has to take into account, that the author worked with the Mition-C version 1.0. Future releases that address this problem are planned and are under development. Currently is for example a new version under construction, which implements streams and introduces by this means more dynamic. So it is in some way too soon to draw definite conclusions regarding the Mitrion-C's limited dynamic.

9.5 Memory management

A known bug in the simulator that is expected to disappear in future releases is that the memory management is not simulated correctly when attempting to read or write concurrently from a same memory bank. The simulator allows concurrent reads and writes despite the fact that only one read or write can occur at a time. Nevertheless, if the programmer is aware of this incorrectness, he can force the memory accesses to occur strictly sequentially by means of passing memory tokens. Doing so is recommended anyway.

A feature that would be useful for the programmer and which may be added in the future, is the handling of memory tokens in a collection. When creating n internal memory banks, the programmer has to write explicitly n-times **_memcreate()**. Unfortunately, this can not be done in a loop, which would potentially decrease the amount of programmed lines by a large factor where a lot of memory banks are created and accessed.

9.6 Appropriate algorithms for Mitrion-C

Algorithms that are very data intensive are not well suited for FPGA implementations. This problem is not restricted to Mitrion, it is rather given by the fact that FPGAs run at relatively low clock frequencies and by the related limited memory bandwidth. A best possible speed-up compared to an ordinary PC implementation is achieved by programs that perform a lot of calculations with few input data. An advantage regarding memory accesses that have FPGAs over CPUs is that FPGAs do not have to deal with cache misses. However, if a CPU succeeds in predicting memory accesses in advance, it may achieve a high cache hit-rate, which makes the CPU advantageous over the FPGA.

A factor, which affects the performance of an FPGA implementation, is the maximum degree of concurrency, which is determined by the algorithm itself (see *Amdahl's law* in chapter 2). Nevertheless, if the procedure can not be implemented in a data-parallel manner because of data dependencies, it may be possible to execute in a pipelined fashion to hide latency. The discussed CORDIC algorithm can be named as an iterative and therefore data-dependent procedure, which could be accelerated through pipelining.

9.7 Dynamic bit-width

The algorithms discussed in this thesis were mostly implemented using standard single precision floating points or single precision integers. Single precision was chosen because of the limited FPGA resources and because of the memory bandwidth. The flip-flop consumption increases often exponentially when choosing double precision values instead. In addition, compared to double precision values, there can be read and written the double amount of single precision values per time frame from and to the external RAM banks. This trade-off between single and double precision may persist in the future. However, if the FPGAs become bigger in terms of available flip-flops, double precision implementations may become more attractive.

A clear advantage of the Mitrion processor over an ordinary PC could be exploited with the ability of utilizing 8-bit unsigned integers in the wavelet transformation (see chapter 7). First of all could the processor read quite a lot of information per time frame, namely 16 values per memory bank and clock cycle. Second, there were no resources wasted for the operations with the 8-bit unsigned integers. The PC on the other hand operates on 32-bit unsigned integers despite the fact that the coefficients only have values in the range between 0 and 255. Though, the wavelet transformation was not very resource consuming. As already mentioned in chapter 7, the spare flip-flops could be used to add an encoding algorithm on the same chip.

9.8 Portability

Although the presented Mitrion-C programs were faster than the ANSI-C programs on the AMD 64 3200+, the Mitrion-C code needs to be modified when run on another platform. Different FPGAs have varying I/O systems and unequal amounts of resources. This is a disadvantage compared to the ANSI-C programs. But in any case are modifications easier and faster made in Mitrion-C than in VHDL.

9.9 Performance results resumed

Summarizing the results of the previous chapters in the table below shows the overall advantage of the Mitrion processor over the AMD 64 3200+ equipped PC.

Algorithm	**Time AMD**	**Time Mitrion**	**speed-up Mitrion over AMD**
Matrix multiplication	$15ms$	$682\mu s$	21.99×
Jacobi linear eq. solver	$48ms$	$7.6ms$	6.32×
Wavelet transformation	$14ms$	$0.614ms$	22.8×
CORDIC	$392\mu s$	$11.19\mu s$	35.63×
Average speed-up			21.67×

A summary of the algorithms, platforms and resource consumptions is found in the table below.

Algorithm	**Platform**	**Precision**	**Flip-Flops**	**RAMs**
Matrix	Virtex II generic	Single	41230 (61%)	64 (44%)
Jacobi	Virtex II generic	Double	39000 (57%)	50 (34%)
Wavelet	SGI RASC v6000	8-bit	23315 (34%)	124 (86%)
CORDIC	Virtex II generic	Single	25818 (38%)	14 (9%)

All of the discussed algorithms were faster on the Mitrion processor than on the AMD, except of the Gaussian elimination, of course. This result meets the expectations and demonstrates the great potential and capabilities of the Mitrion processor. However, one has to incorporate the fact that the systems compared are not only of different architectures, but also of very different price segments. While about 1000 Euros buy an up-to-date AMD 64 equipped PC, are the costs of an FPGA based supercomputer at least 10 times higher.

Chapter 10

Conclusions

In the previous chapters, the author discussed Mitrion-C implementations of a range of mathematical functions and compared them with ANSI-C implementations for a sequential processor. It was expected that the Mitrion processor is about 20 times faster than the sequential processor regarding these functions. Apart from one algorithm, which failed to be implemented in Mitrion-C, were all other algorithms on the Mitrion processor faster by an average factor of almost 22. Hence, the promise was fulfilled with reservation of the mentioned exception.

New features that will soon be added to the Mitrion-C language address the dynamic problem, which prevented a successful implementation of the remaining algorithm. It is therefore too soon to draw final conclusions regarding the lacking dynamic of Mitrion-C.

Mitrion-C programming introduces a very high abstraction layer to hardware programming. The language together with the Mitrion SDK makes the hardware design process easier and faster than ever.

In comparison with the sequential processor is the Mitrion processor advantageous when the algorithms are not very data intensive and not highly dynamic.

Current reconfigurable hardware seems to be well mated for supercomputing. Certain classes of algorithms profit amazingly from customized hardware. Mitrion adds even more advantages to the reconfigurable supercomputing world, in that the well designed Mitrion-C programming language reduces time of production and debugging.

The author was in general very happy with Mitrion and reckons great potential for future releases together with new and even more sophisticated FPGA technologies.

Chapter 11

Future work

The previous chapters demonstrated the great potential and capabilities of the Mitrion platform. The discussed algorithms were successfully implemented and all of them beat the AMD 64 3200+ equipped PC. Nevertheless, there is always amelioration and extension possible. The author makes some suggestions in this chapter.

11.1 Dynamic of Mitrion-C

The Mitrion-C programs were developed using version 1.0 of the Mitrion SDK. Future releases with more dynamic support would be welcome. Solving systems of linear equations with one of the direct methods such as Gaussian elimination or LU-factorization would then become possible to implement efficiently. However, the author does not exclude the existence of a direct method, which could avoid the dynamic loop-length problem. More time could be invested in finding an appropriate variant of the mentioned algorithms, which would be suited for the current state of the Mitrion-C language. To run Gaussian elimination with partial pivoting on the Mitrion processor would be of special interest because there exists a broad benchmark on [20], which compares various computer systems ranging from workstations to sophisticated clusters regarding this algorithm.

11.2 JPEG 2000

As an extension to the discrete Haar wavelet transformation presented in chapter 7, the author already suggested to add an appropriate data encoding algorithm such as embedded zero tree coding or Huffman coding. The wavelet transformation as well as the encoding could be done according to the JPEG 2000 standards, which would allow the Mitrion processor to be compared with various implementations made in other projects.

11.3 Memory simulation

As already mentioned, the Mitrion simulator does not correctly simulate concurrent memory accesses. The bug in the simulator can indeed be handled by passing memory tokens, but it would be eligible to be able to rely on the simulator concerning this subject.

11.4 Memory token handling

In chapter 4, the discussion was about matrix multiplication. The algorithm multiplies a quite large number of elements in parallel. To be able to access a number of elements of either matrix simultaneously, the entries have to be stored at different memory locations. It would be desirable to be able to express the reading of all elements by the means of a **foreach** loop instead of typing for each element a **_memread()**.

11.5 Resource usage optimization

The discussion in chapter 9 brought up the actuality that all function calls in Mitrion-C are made in-line at compile time. This fact averts to implement recursive algorithms to any desired level because of the fast increasing resource usage. Optimizations might be possible that would allow the programmer to explicitly declare functions in-line or not.

11.6 Comparison with OpenMP and MPI

As a suggestion for future projects, the author proposes to compare the Mitrion processor as well as the programming language Mitrion-C with the well known standards OpenMP [21] and MPI [22] and the respecive systems on which such programs can be run. Comparing to the fine-grained FPGA based platform Mitrion are both standards considered coarse-grained. MPI (Message Passing Interface) is a standard for message passing, which can be embedded in ANSI-C programs. OpenMP can as well be integrated in ANSI-C programs, but it is a standard for shared memory architectures.

11.7 Financial aspect

In the context of a comparison of the Mitrion processor with workstation clusters using OpenMP or MPI, an interesting point would be to analyze the financial aspect. Since FPGAs are very economical in energy consumption, the Mitrion processor might not only be advantageous in terms of programmability but also in terms of money.

Bibliography

[1] Eduardo Sanchez. An introduction to digital systems. Technical report, Swiss Federal Institute of Technology Lausanne, EPFL, 1998.

[2] Henrik Abelsson. Evaluation of a parallel reconfigurable architecture implemented in fpgas for supercomputing applications. Master's thesis, University of Linköping, 2005.

[3] Hello supercomputing world. Mitrionics advertising brochure.

[4] ETH Zuerich Prof. Matthias Troyer. Computational physics, 2005/2006.

[5] http://en.wikipedia.org/wiki/unsolved_problems_in_computer_science/.

[6] http://mayet.som.yale.edu/coopetition/vn.html.

[7] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Parallel Computing*. Addison Wesley, second edition, 2003.

[8] http://en.wikipedia.org/wiki/amdahl's_law.

[9] Stefan Möhl. The mitrion-c programming language.

[10] http://www.eda.org/vasg/.

[11] http://www.verilog.com.

[12] http://www.eda.org/vasg/.

[13] Stefan Möhl. Mitrion-c presentation.

[14] Gerald R. Morris and Viktor K. Prasanna. An fpga-based floating-point jacobi iterative solver. Technical report, Department of Electrical Engineering, University of Southern California, 2005.

[15] Wim Sweldens Peter Schroeder. Wavelets in computer graphics. Technical report, California Institute of Technology and Lucent Technologies Bell Laboratories, 1996.

[16] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Wavelet transforms that map integers to integers. *Appl. Comput. Harmon. Anal.*, 1998.

[17] Khalid Sayood. *Introduction to Data Compression.* Morgan Kaufmann Publishers, second edition, 2000.

[18] Daniel Bachofen. Fpga wavelet transformation für bildübertragung. Master's thesis, Hochschule für Technik, Wirtschaft und soziale Arbeit, St. Gallen, Switzerland, 2001.

[19] http://www.dspguru.com/info/faqs/cordic.htm.

[20] http://www.top500.org.

[21] http://www.openmp.org/.

[22] http://www.mpi-forum.org/.

[23] http://www.processing.org.

Index

Appendix A

Graphical debugger

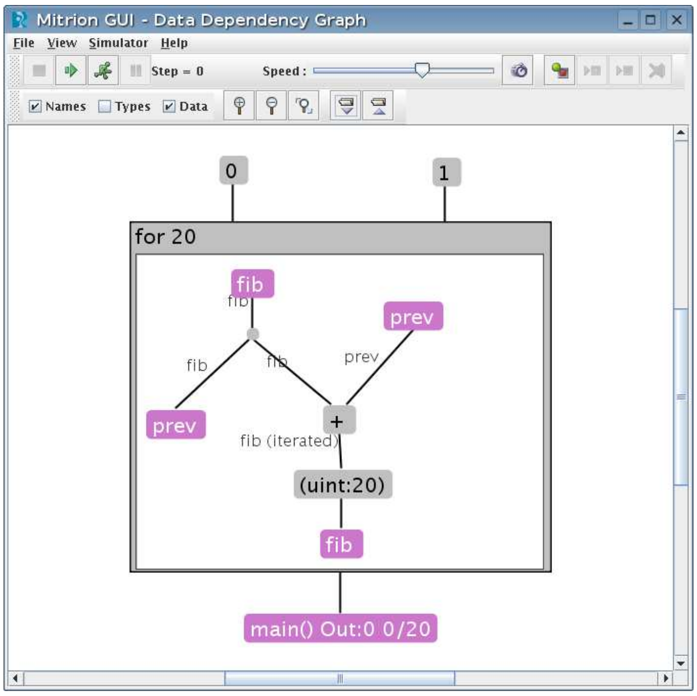

Figure A.1: The graphical debugger

Appendix B

Wavelet transformation

B.1 Source code

B.1.1 Mitrion-C program

```
Mitrion-C 1.0;

#define memtype bits:128
#define EVEN 0
#define ODD 1


(uint:8[8], uint:8[8]) filter (uint:8[8][2] buffer)
{
  (Lo, Hi) = foreach(pair in buffer) {
    uint:8 low = (pair[EVEN] + pair[ODD]) / 2;
    uint:8 high = pair[EVEN] - pair[ODD];
  } (low, high);
} (Lo, Hi);

(uint:8[8], uint:8[8]) filter2 (uint:8[2][8] buffer)
{
  (Lo, Hi) = foreach(i in [0..7]) {
    uint:8 low = (buffer[EVEN][i] + buffer[ODD][i]) / 2;
    uint:8 high = buffer[EVEN][i] - buffer[ODD][i];
  } (low, high);
} (Lo, Hi);


(uint:8[8][2]) getpixels (bits:128 buffer)
{
  bits:8[8][2] bitvect = buffer;

  (pixels) = foreach(pair in bitvect) {
    (pixels) = foreach(e in pair) {
      uint:8 pixel = e;
    } (pixel);
  } (pixels);

} (pixels);


(bits:64) makebits64 (uint:8[8] buffer)
{
  (tmp) = foreach(e in buffer) {
    bits:8 byte = e;
    bits:64 tmp = byte;
  } (tmp);

  bits:64 a = tmp[0] << 56;
  bits:64 b = tmp[1] << 48;
  bits:64 c = tmp[2] << 40;
  bits:64 d = tmp[3] << 32;
  bits:64 e = tmp[4] << 24;
  bits:64 f = tmp[5] << 16;
  bits:64 g = tmp[6] << 8;
  bits:64 h = tmp[7];
  bits:64 buffer0 = a | b | c | d | e | f | g | h;

} (buffer0);


(bits:128) makebits128 (bits:64 a, bits:64 b)
{
  bits:128 a0 = a;
  bits:128 b0 = b << 64;
  bits:128 buffer = a0 | b0;
} (buffer);



(mem memtype [0x20000],
 mem memtype [0x20000]) main (mem memtype [0x20000] Am,
                              mem memtype [0x20000] Bm)

{
  Cm = _memcreate(mem memtype[0x3000] Cm_end);

  (Cm0, LoHi_bits, HiLo_bits, HiHi_bits) = foreach(row in <0..191>) {

    /* Level 1 Rowwise */
    (LoLo, LoHi_bits,
     HiLo_bits, HiHi_bits) = foreach(col in <0..15>) {

       (Lo, Hi) = foreach(k in <0..1>) {
         (Lo, Hi) = foreach(l in <0..1>) {
           (Lo, Hi) = foreach(i in <0..1>) {
             (Hi, Lo) = foreach(j in <0..1>) {

               uint:14 memloc = row*256+col*4 + l*128 + k*2 + j*64 + i;
               memtype buffer = _memread(Am, memloc);

               (pixels) =  getpixels(buffer);
               (Lo, Hi) = filter(pixels);

             } (Lo, Hi);
           } (Lo, Hi);
         } (Lo, Hi);
       } (Lo, Hi);

       tmpLo = reshape(Lo, <8><2>[8]);
       tmpHi = reshape(Hi, <8><2>[8]);

       Lo_re = reformat(tmpLo, <8>[2][8]);
       Hi_re = reformat(tmpHi, <8>[2][8]);
```

```
   /* Level 1 Columnwise high pass */
   (HiHi_bits, HiLo_bits) = foreach(Hi in Hi_re)
   {
     (HiLo, HiHi) = filter2(Hi);
     HiHi_bits = makebits64 (HiHi);
     HiLo_bits = makebits64 (HiLo);
   } (HiHi_bits, HiLo_bits);


   /* Level 1 Columnwise low pass  */
   (LoLo, LoHi) = foreach(Lo in Lo_re) {
     (LoLo, LoHi) = filter2(Lo);
   } (LoLo, LoHi);

   (LoHi_bits) = foreach(lohi in LoHi) {
     LoHi_bits = makebits64(lohi);
   } (LoHi_bits);

 } (LoLo, LoHi_bits, HiLo_bits, HiHi_bits);



/* Level 2 row wise */
LoLo_tmp = reshape(LoLo, <64><2>[8]);
LoLo_tmp0 = reformat(LoLo_tmp, <64>[2][8]);
LoLo_re = reshape(LoLo_tmp0, <64>[8][2]);

(LoLoL, LoLoH) = foreach(LoLo in LoLo_re) {
  (LoLoL, LoLoH) = filter(LoLo);
} (LoLoL, LoLoH);


LoLoL_tmp = reshape(LoLoL, <32><2>[8]);
LoLoH_tmp = reshape(LoLoH, <32><2>[8]);

 LoLoL_re = reformat(LoLoL_tmp, <32>[2][8]);
 LoLoH_re = reformat(LoLoH_tmp, <32>[2][8]);

 /* Level 2 column wise high pass */
 (LoLoHH_bits, LoLoHL_bits) = foreach(hi in LoLoH_re) {
   (HiLo, HiHi) = filter2(hi);
   HiHi_bits = makebits64 (HiHi);
   HiLo_bits = makebits64 (HiLo);
 } (HiHi_bits, HiLo_bits);

 /* Level 2 column wise low pass */
 (LoLoLL_bits, LoLoLH_bits) = foreach(lo in LoLoL_re) {
   (LoLo, LoHi) = filter2(lo);
   LoHi_bits = makebits64 (LoHi);
   LoLo_bits = makebits64 (LoLo);
 } (LoHi_bits, LoLo_bits);

 LLHH_tmp = reshape(LoLoHH_bits, <16><2>);
 LLHL_tmp = reshape(LoLoHL_bits, <16><2>);
 LLLH_tmp = reshape(LoLoLH_bits, <16><2>);
 LLLL_tmp = reshape(LoLoLL_bits, <16><2>);

 LLHH_re = reformat(LLHH_tmp, <16>[2]);
 LLHL_re = reformat(LLHL_tmp, <16>[2]);
 LLLH_re = reformat(LLLH_tmp, <16>[2]);
 LLLL_re = reformat(LLLL_tmp, <16>[2]);
```

```
   (LLll_128, LLlh_128,
    LLhl_128, LLhh_128) = foreach(llll, lllh,
                                  llhl, llhh in
                                  LLLL_re, LLLH_re,
                                  LLHL_re, LLHH_re)
  {
    llll_128 = makebits128(llll[0], llll[1]);
    lllh_128 = makebits128(lllh[0], lllh[1]);
    llhl_128 = makebits128(llhl[0], llhl[1]);
    llhh_128 = makebits128(llhh[0], llhh[1]);
  } (llll_128, lllh_128, llhl_128, llhh_128);


  /* Store result for second pass in temp location */
  (Cm3) = foreach (ll, lh, hl, hh, index in
                   LLll_128, LLlh_128,
                   LLhl_128, LLhh_128, <0..15>)
  {
    uint:10 memloc = row*32 + index;
    Cm0 = _memwrite(Cm, memloc, ll);
    Cm1 = _memwrite(Cm0, memloc+16, hl);
    Cm2 = _memwrite(Cm1, memloc+6144, lh);
    Cm3 = _memwrite(Cm2, memloc+6160, hh);

  } (Cm3);


  LoHi_tmp = reshape(LoHi_bits, <64><2>);
  HiLo_tmp = reshape(HiLo_bits, <64><2>);
  HiHi_tmp = reshape(HiHi_bits, <64><2>);

  LoHi_re = reformat(LoHi_tmp, <64>[2]);
  HiLo_re = reformat(HiLo_tmp, <64>[2]);
  HiHi_re = reformat(HiHi_tmp, <64>[2]);

  (LoHi_128, HiLo_128, HiHi_128) = foreach(lh, hl, hh in
                                     LoHi_re, HiLo_re, HiHi_re)
  {
    lh128 = makebits128(lh[0], lh[1]);
    hl128 = makebits128(hl[0], hl[1]);
    hh128 = makebits128(hh[0], hh[1]);

  }(lh128, hl128, hh128);

} (Cm3, LoHi_128, HiLo_128, HiHi_128);



LoHi_re = reshape(LoHi_bits, <192><16><2><2>);
HiLo_re = reshape(HiLo_bits, <192><16><2><2>);
HiHi_re = reshape(HiHi_bits, <192><16><2><2>);

(Bm3) = foreach(lh, hl, hh, k in
                LoHi_re, HiLo_re, HiHi_re, <0..191>) {
  (Bm3) = foreach(lh1, hl1, hh1 , l in
                  lh, hl, hh, <0..15>) {
    (Bm3) = foreach(lh2, hl2, hh2, i in
                    lh1, hl1, hh1, <0..1>) {
      (Bm3) = foreach(lh3, hl3, hh3, j in
                      lh2, hl2, hh2, <0..1>) {
        int:14 memloc = k*128 + l*2 + j*64 + i;
        Bm1 = _memwrite(Bm,  memloc + 32 ,hl3);
```

```
          Bm2 = _memwrite(Bm1,  memloc + 24576 ,lh3);
          Bm3 = _memwrite(Bm2,  memloc + 24608 ,hh3);
        } (Bm3);
      } (Bm3);
    } (Bm3);
  } (Bm3);


  Bm_tmp = _wait(_wait(_wait(_wait(Bm3))));
  Cm_tmp = _wait(_wait(Cm0));


  /* Write back buffer of second pass */
  (Bm_e, Cm_e) = foreach(i in <0..383>) {
    (Bm0, Cm0) = foreach(j in <0..31>) {
      (buffer, Cm0) = _memread(Cm_tmp, i*32+j);
      Bm0 = _memwrite(Bm_tmp, i*64+j, buffer);
    } (Bm0, Cm0);
  } (Bm0, Cm0);


  Bm_last = _wait(_wait(Bm_e));
  Cm_end = _wait(_wait(Cm_e));

} (Am,  Bm_last);
```

B.1.2 Host program

```
#include <stdlib.h>
#include <stdio.h>

#include "mithal.h"
#include "mithal_gen.h"

#define WORD unsigned char
#define NUM_WORDS 786432
#define COLS 1024
#define ROWS 768

int main(int argc, char** argv)
{
  FPGA *f;
  Processor *p;
  STATUS s;
  WORD *mem1, *mem2;
  WORD next;
  int i, j;
  FILE *input, *output;

  f = mitrion_fpga_allocate("localhost:60000");
  if (f == NULL) {
    fprintf(stderr, "Could not find the FPGA.\n");
    return 22;
  }

  p = mitrion_processor_create("haar.2D.L2.mitc");
  if (p == NULL) {
    fprintf(stderr, "Coult not create the processor.\n");
    return 23;
  }

  s = mitrion_fpga_load_processor(f, p);
  if (s != OK) {
    fprintf(stderr, "Could not load the processor
                     on the FPGA.\n");
    return 24;
  }

  mem1 = (WORD*)mitrion_processor_reg_buffer(p, "Am", NULL,
                         NUM_WORDS*sizeof(WORD), WRITE_DATA);

  mem2 = (WORD*)mitrion_processor_reg_buffer(p, "Bm", NULL,
                         NUM_WORDS*sizeof(WORD), READ_DATA);


  // Prepare data (read col-wise and write row-wise)
  input = fopen("snowboarder.bin", "rb");
  if (input == 0) {
    printf("Unable to open input file.\n");
    exit -1;
  }
  for (i=0; i<COLS; i++) {
    for (j=0; j<ROWS; j++) {
      next = fgetc(input);
      mem1[j*COLS+i] = next;
    }
  }
  fclose(input);
  // run processor
  mitrion_processor_run(p);
  mitrion_processor_wait(p);


  // Write back result (read row-wise and write col-wise)
  output = fopen("snowboarder.transformed.bin", "wb");
  if (output == 0) {
    printf("Unable to open output file.\n");
    exit -1;
  }
  for (i=0; i<COLS; i++) {
    for (j=0; j<ROWS; j++) {
      next = mem2[j*COLS+i];
      fputc(next, output);
    }
  }
  fclose(output);


  mitrion_fpga_close(f);
  return 0;
}
```

B.2 A visual demonstration

B.2.1 The original image

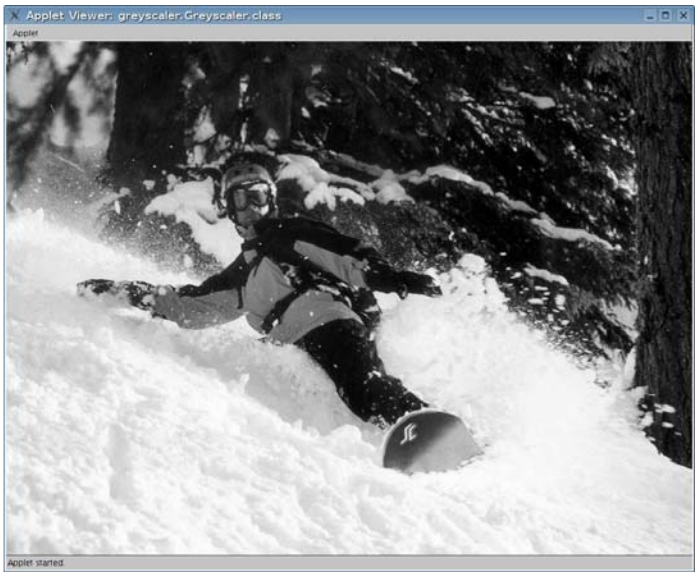

Figure B.1: Original image

The original image was a colored JPEG image, which the author turned into a gray-scale image by the means of *Processsing* [23]. The gray-scale image could then be written pixel-wise in a file using Java and Processing API's. After the wavelet transformation, the images could be read from the file and be displayed again.

B.2.2 The transformed images

As mentioned in chapter 7, the image is transformed into approximation and detail coefficients. The approximation coefficients can be seen as a coarse representation of the original image. If the image is transformed once horizontally and once vertically, it looks as depicted in B.2. The coarse representation (the LL-coefficients) are in the upper left part. The other four parts are the detail coefficients (LH, HL, HH). It does not make much sens to visualize them as such, but they are needed to add details, if the image is back transformed. The detail coefficients are mostly close to black (intensity 0) or close to white (intensity 255) because they represent differences between neighboring pixels of the original image. Threshold the detail coefficients (lossy transformation) could turn many of the white pixels into black ones, which in turn would reduce the entropy even more and hence augment the achievable compression rate. An advantage of wavelet-transformed images is that if they are to be displayed, one can make very fast previews.

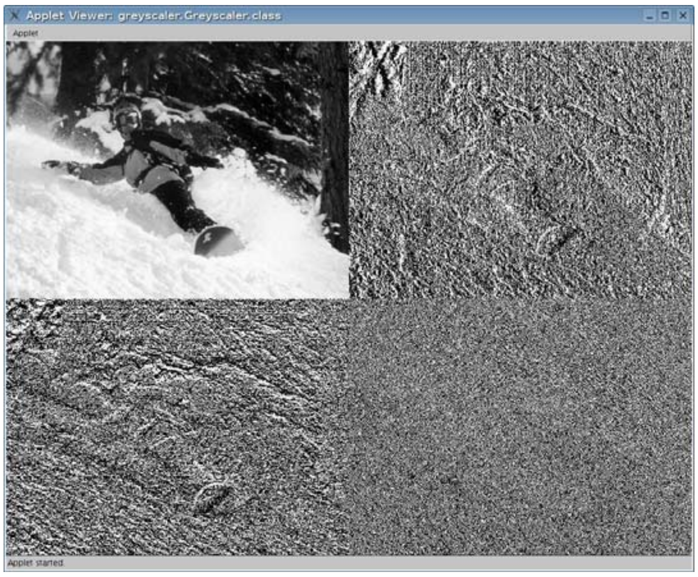

Figure B.2: Level 1 transformation (2 dimensional)

Figure B.3 shows the image after the second tranformation level. The second transformation level transforms the LL coefficients of the first transformation level once again.

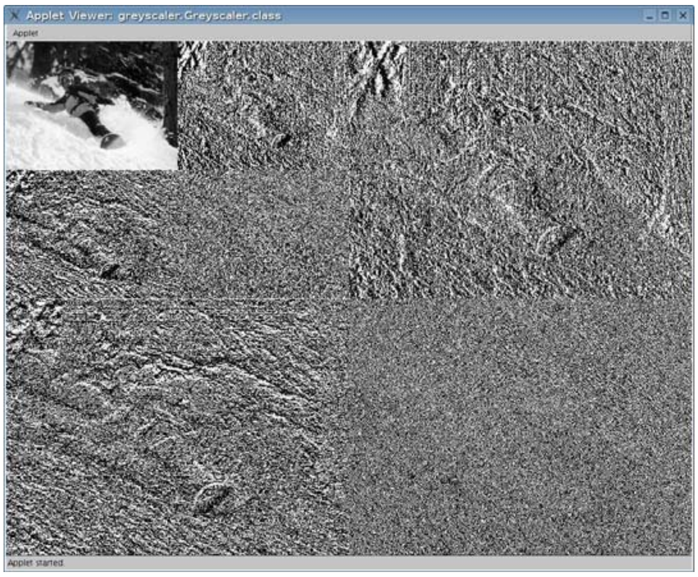

Figure B.3: Level 2 transformation (2 dimensional)

B.3 Dependency graph

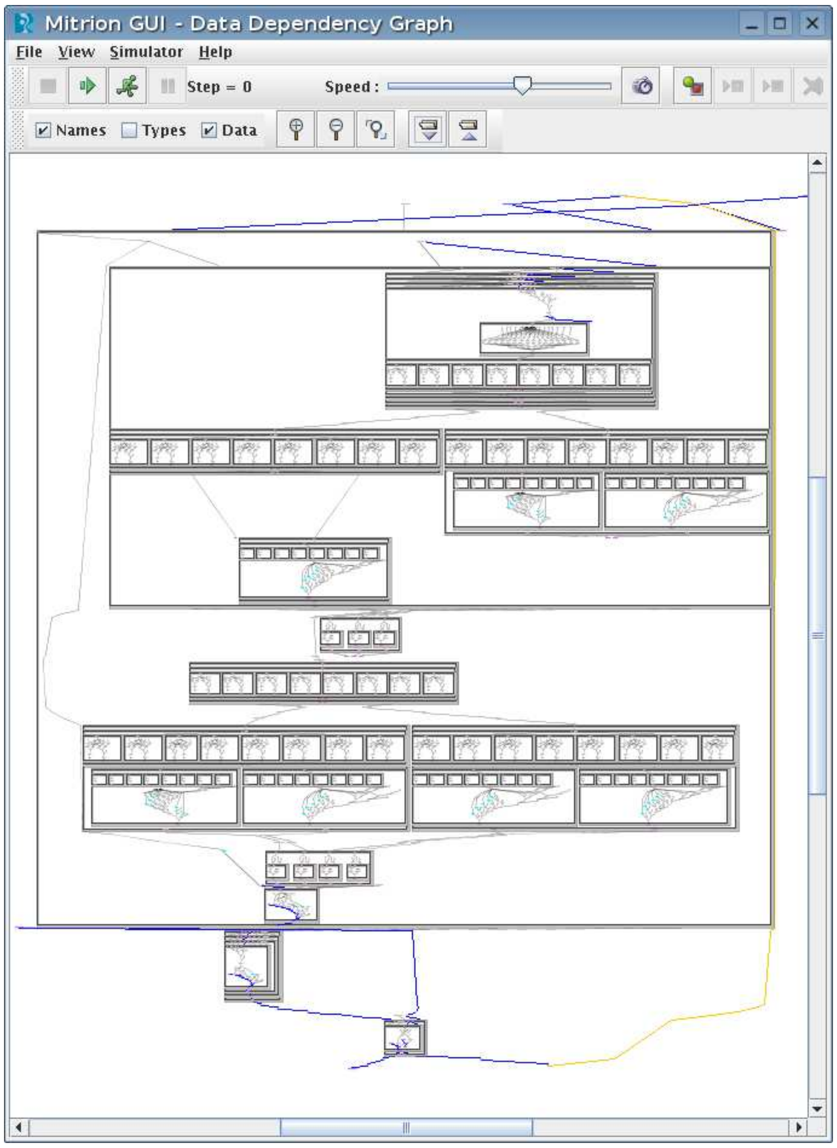

Figure B.4: Dependency graph of the wavelet transformation