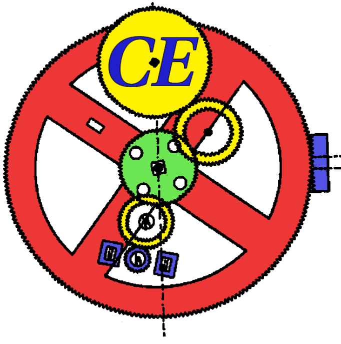


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

IWEX

A.Thomas

CE-MS-2009-29

Abstract

IWEX stands for Invers Wavefield Extrapolation and is commonly used in seismic exploration. Niels Pörtzgen[15] showed that this technique can also be used for inspection of welds. This thesis describes two domains in which the IWEX algorithm can be calculated. The time domain which has a quantization problem and the frequency domain which is computation intensive. The computation requirements run up to 5.5 TFLOPS(typical) for the frequency domain. To decrease the execution time of the frequency domain algorithm three platforms are explored: GPUs, the Cell processor and FPGAs. GPUs are originally designed for the game market to render 3D images, but over the last couple of years the GPU has become much more usable for general purpose computing. One of the latest GPUs has 240 scalar processors and has a theoretical 933 GFLOPS[14]. The Cell processor has eight light weight cores and one power processor on one chip. With a theoretical 204.8 GFLOPS also a powerful device[5]. The third platform is the FPGA. The floating point performance of the FPGA is lower than the other two platforms but if fix-point computations are used the FPGA is expected to execute the IWEX algorithm the quickest(4.9 s) of the three platforms. The GPU is expected to execute the algorithm in 14.5 seconds, and the Cell processor in 26.8 seconds. We implemented the algorithm on the GPU platform with the result that one image takes 15.1 seconds which indicates 184 GFLOPS. This implies an efficiency of 19.7%. But the implementation has a speedup of 20 compared to a quad-core CPU based implementation.

Faculty of Electrical Engineering, Mathematics and Computer Science

IWEX
Mapping on current high performance architectures

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

A.Thomas
born in Meppel, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

IWEX

by A.Thomas

Abstract

IWEX stands for Invers Wavefield Extrapolation and is commonly used in seismic exploration. Niels Pörtzgen[15] showed that this technique can also be used for inspection of welds. This thesis describes two domains in which the IWEX algorithm can be calculated. The time domain which has a quantization problem and the frequency domain which is computation intensive. The computation requirements run up to 5.5 TFLOPS(typical) for the frequency domain. To decrease the execution time of the frequency domain algorithm three platforms are explored: GPUs, the Cell processor and FPGAs. GPUs are originally designed for the game market to render 3D images, but over the last couple of years the GPU has become much more usable for general purpose computing. One of the latest GPUs has 240 scalar processors and has a theoretical 933 GFLOPS[14]. The Cell processor has eight light weight cores and one power processor on one chip. With a theoretical 204.8 GFLOPS also a powerful device[5]. The third platform is the FPGA. The floating point performance of the FPGA is lower than the other two platforms but if fix-point computations are used the FPGA is expected to execute the IWEX algorithm the quickest(4.9 s) of the three platforms. The GPU is expected to execute the algorithm in 14.5 seconds, and the Cell processor in 26.8 seconds. We implemented the algorithm on the GPU platform with the result that one image takes 15.1 seconds which indicates 184 GFLOPS. This implies an efficiency of 19.7%. But the implementation has a speedup of 20 compared to a quad-core CPU based implementation.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2009-29

Committee Members :

Advisor:	Arjan van Genderen, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Stephan Wong, CE, TU Delft
Member:	Emile Hendriks, I&CT, TU Delft
Member:	Cees van Teylingen, Chess

Contents

List of Acronyms **vi**

List of Figures **vii**

List of Tables **ix**

Acknowledgments **xi**

1 Introduction **1**
1.1 IWEX . . . 1
1.2 Thesis organization . . . 1

2 Algorithm **3**
2.1 Time domain . . . 4
2.2 Frequency domain . . . 6

3 Parameters and complexity of the algorithm **9**
3.1 Parameters . . . 9
3.2 Time domain . . . 9
3.3 Frequency domain . . . 10
3.4 Number of samples . . . 10
3.5 Single precision vs double precision floating point . . . 11
3.6 Conclusions . . . 12

4 Platform exploration **13**
4.1 GPU . . . 13
4.1.1 Architecture . . . 13
4.1.2 Related research . . . 15
4.1.3 Mapping of the algorithm . . . 16
4.1.4 Conclusions . . . 19
4.2 FPGA . . . 19
4.2.1 Floating point performance with respect to the frequency domain . 20
4.2.2 Fix-point calculation . . . 21
4.2.3 Mapping of the algorithm . . . 21
4.2.4 Conclusions . . . 26
4.3 Cell . . . 26
4.3.1 Architecture . . . 26
4.3.2 Mapping of the algorithm . . . 28
4.3.3 Conclusions . . . 29
4.4 Conclusions . . . 30

5 Implementation 31
5.1 Implementation time domain . . . 31
5.1.1 Original . . . 31
5.1.2 Pre-calculation . . . 32
5.2 Implementation frequency domain . . . 35
5.2.1 Original . . . 35
5.2.2 Pre-calculation . . . 38
5.3 Optimizations . . . 39

6 Results 41
6.1 Time domain . . . 41
6.1.1 Original algorithm . . . 41
6.1.2 Pre-calculation . . . 43
6.2 Frequency domain . . . 43

7 Conclusions and future work 51
7.1 Conclusions . . . 51
7.2 Remarks . . . 51
7.3 Future work . . . 52

Bibliography 54

A Time domain algorithm 55

B Complex numbers 57

C Time domain implementation code 59
C.1 CPU code . . . 59

D Frequency domain algorithm 61

E Frequency domain algorithm code 63
E.1 CPU code . . . 63

F MATLAB commands introduction 65

G Prototype 67
G.1 IWEX algorithm: Memory and computation requirements . . . 68
G.1.1 Computation requirements . . . 68
G.1.2 Memory requirements . . . 69

H Single precision versus double precision 71

List of Acronyms

ALU	Arithmetic Logic Unit.
BLAS	Basic Linear Algebra Subprograms.
CPU	Central Processing Unit.
d_element	heart to heart distance between the piëzo elements.
DMA	Direct Memory Access.
DP	Double Precision.
DSP	Digital Signal Processor.
EIB	Element Interconnect Bus.
FF	Flip-Flop.
FLOP	Floating Point OPeration.
FLOPS	FLoating point Operations Per Second.
FPGA	Field Programmable Gate Array.
GDDR3	Graphics Double Data Rate 3.
GFLOPS	Giga(10^9) FLoating point Operations Per Second.
GPU	Graphics Processing Unit.
GS	Geometric Spreading.
HDL	Hardware Description Language.
IWEX	Invers Wavefield EXtrapolation.
LUT	LookUp Table.
MA	Multiply-Add.
MB	MegaByte.
N_el	Number of piëzo elements.
N_SAMPLES	Number of samples.
PPE	Power Processing Element.
ROI	Region Of Interests.

SIMD	Single Instruction Multiple Data.
SIMT	Single Instruction Multiple Thread.
SP	Single Precision.
SPE	Synergistic Processing Element.

List of Figures

1.1 An IWEX picture applied on steel 2

2.1 Visual indication of the parameters. 4
2.2 Defect in metal reflects ultra soon signals. 4
2.3 Grid defined in the metal. 5
2.4 Example IWEX picture of time domain 6
2.5 Example IWEX picture of frequency domain 7

3.1 Number of pixels divided in fault percentages intervals 12

4.1 CPU vs GPU [14] 14
4.2 Architecture of a GPU [14] 14
4.3 Hardware needed for the GPU solution 16
4.4 FPGA block diagram time domain 22
4.5 Block diagram for calculating ti 23
4.6 Block diagram for calculating in frequency domain 25
4.7 Architecture of a Cell [5] 27
4.8 Architecture of a SPE [5] 28

5.1 Output matrix divided into blocks (this case block size of 2) 32
5.2 Vector distance of a)transmitter and receiver 1; b)transmitter and receiver 2; c)transmitter and receiver 3. With indexes to the original matrix . . . 33
5.3 Sub matrices multiplication with the first and second iteration (r_itt) . . . 36

6.1 Time and FLOPs versus N_el of the original time domain 42
6.2 Time and FLOPs versus ROI of the original time domain 42
6.3 Original time domain algorithm versus resolution 43
6.4 Original time domain algorithm versus pre-calculated 44
6.5 Original frequency domain algorithm versus pre-calculated 44
6.6 Execution time versus number of elements 45
6.7 Execution time versus the region of interest 45
6.8 Execution time versus the resolution 46
6.9 Monitor overhead for the time and frequency domain 46
6.10 Execution time 1 GPU versus 4 GPU's 47
6.11 CPU versus GPU 47
6.12 Time domain versus frequency domain 48
6.13 System performance using time domain with 64 piëzo elements 48

G.1 IWEX system overview 67
G.2 Organization of the chipset [3] 70

H.1 Results of single precision output vs. double precision output 72

List of Tables

2.1 Parameters needed by the IWEX algorithm see figure 2.1 3

3.1 Variables 9
3.2 Computational requirements of the time domain 10
3.3 Computational requirements of the frequency domain 11

4.1 Properties of a multiprocessor 15
4.2 Devices used for estimation 20
4.3 Operations and their resource usage 20
4.4 Overview of the properties of the different platforms 30

6.1 The values used for the benchmarks if not specified otherwise 41

G.1 Sizes of matrices dependencies 69

H.1 (P)SNR dependent on the picture and resolution 73

Acknowledgments

I want to thank a number of people who have spent their time helping me with this thesis project. First thanks to Roger van Schie, Teun van Kuppeveld and Cees van Teylingen for your guidance and input. Thanks to Arjan van Genderen who guided me to the final result. Thanks also to my girlfriend Sanne who supported me during the project. Finally thanks to all my family and friends who were interested in this thesis project while most of you do not understand it.

A.Thomas
Delft, The Netherlands
October 30, 2009

1 Introduction

1.1 IWEX

Invers Wavefield EXtrapolation (IWEX) is the name of a technique commonly used in seismic exploration, to create an image of ground layers and structures, e.g. in order to find oil and gas. In a study performed by Niels Pörtzgen[15] this technique is now applied in the field of ultrasonic inspection of welds in steel. The expectation is that this method will lead to easier understanding of the data due to better displaying with respect to current techniques. In an IWEX measurement (applied to steel) an array of ultrasonic piëzo elements are excited one after another. After every excitation, all elements in the array record the ultrasonic signals resulting from echos in the material. The signals are sampled, digitized and stored. Next, the wave field at each position in the material is calculated from the received data using a discrete Rayleigh II integral, resulting in a complete image of the material. Figure 1.1 shows an example of the resulting image.

The first implementation of the IWEX system is based on a Intel Quad-core platform. This platform is sufficient for a small amount(32) of piëzo elements. To increase the resolution of the final image more piëzo elements are required, and at that point the Intel platform does not have enough computation power. The algorithm takes to much time to be useful in a practical situation. Appendix G describes the prototype.

The problem statement for this thesis project is therefore to investigate which architecture is the most suited to run the IWEX algorithm and to give an indication of the performance on that particular architecture. The suitability is not just the architecture that runs the algorithm the quickest but it has to be able to acquire the data and display the result.

1.2 Thesis organization

In the next chapter the original algorithm is explained. Chapter 3 sets the requirements that is needed by the system. In Chapter 4 platforms suitable for this algorithm are elaborated on, as well as the advantages and disadvantages of the specific platforms with respect to the algorithm. Chapter 5 describes the implementation, the problems encountered and the solutions to them. In Chapter 6 the results of the implementation tests are described and compared to the theoretical limitations and the original implementation. This thesis is concluded in Chapter 7 where also recommendations for future work can be found.

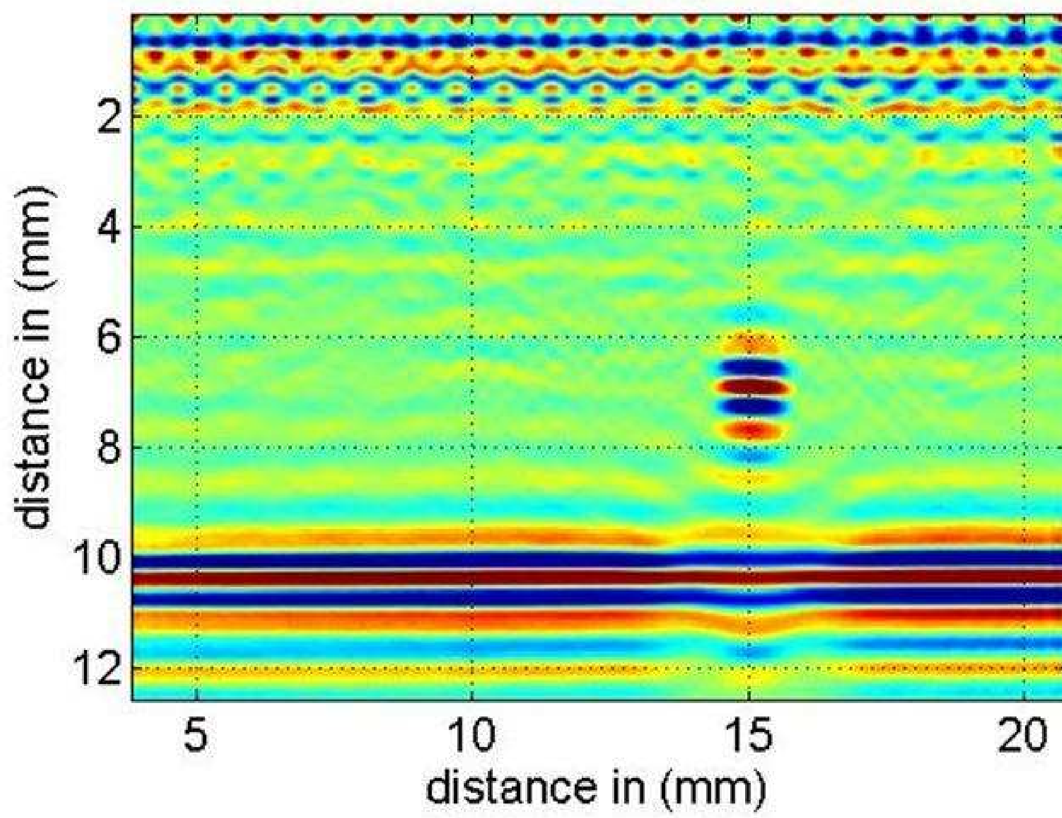


Figure 1.1: An IWEX picture applied on steel

2 Algorithm

For the inspection of metal welds, piëzo elements are used to transmit waves into the metal and also to receive waves from the metal. In figure 2.2 the piëzo elements are on the top in the x direction. Δx is the heart to heart distance between the piëzo elements (d_element). The z direction is a cross section of the metal below the piëzo elements. The curve in the x,z space represents a defect or the bottom of the metal. To gather the required data for the IWEX algorithm, the first element starts to send a signal. This signal travels through the metal (down direction) until it hits a defect or the bottom of the metal. At that point the signal reflects and travels backwards (up direction). Right after the signal is sent by a piëzo element, all the available elements start to receive, and if there was a reflection of the signal, this reflection is received by the elements. To complete the necessary data, all elements have to send one after the other. This means that the received data consists of a 3-dimensional array with the dimensions of the number of available piëzo elements (N_el)(receiving), the number of samples taken to sample the signal, and the number of piëzo elements for transmitting. Note that there is only one set of piëzo elements, but the algorithm makes a distinction between the transmitting and receiving elements.

The algorithm is based on Kirchhoff integrals about wave propagation and extrapolations. These integrals can be formulated in three domains, time, frequency and wave number domain. All three domains have advantages and disadvantages. The wavenum-

Name	Description
d_element	Heart to heart distance of the piëzo elements.
d_wall	Thickness of the metal under examination.
cl_1	Propagating velocity of the signal in metal.
N_el	Number of piëzo elements (N_el).
i_f_start	First frequency to be used by the algorithm.
i_f_end	Last frequency to be used by the algorithm.
xi_ROI_start	x distance expressed in elements of the start of the Region Of Interest.
xi_ROI_end	x distance expressed in elements of the end of the Region Of Interest.
zi_res	Resolution expressed in number of pixels between two piëzo elements(x direction).
xi_res	Resolution expressed in number of pixels between two piëzo elements(z direction).
Number of samples	Number of samples that have to be taken at acquisition.

Table 2.1: Parameters needed by the IWEX algorithm see figure 2.1

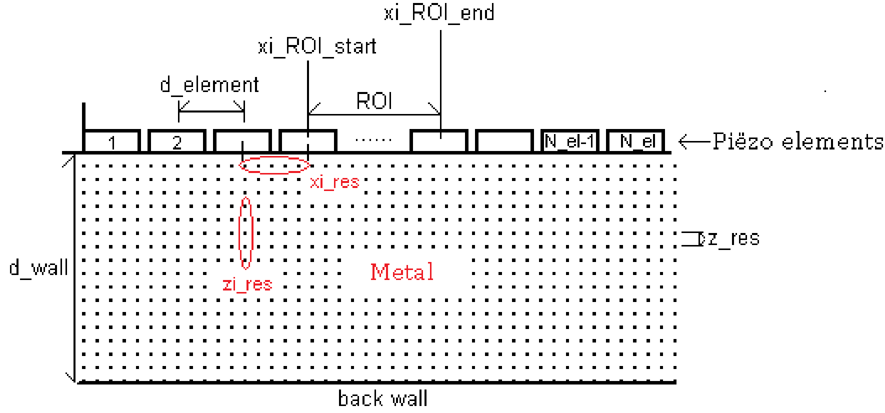


Figure 2.1: Visual indication of the parameters.

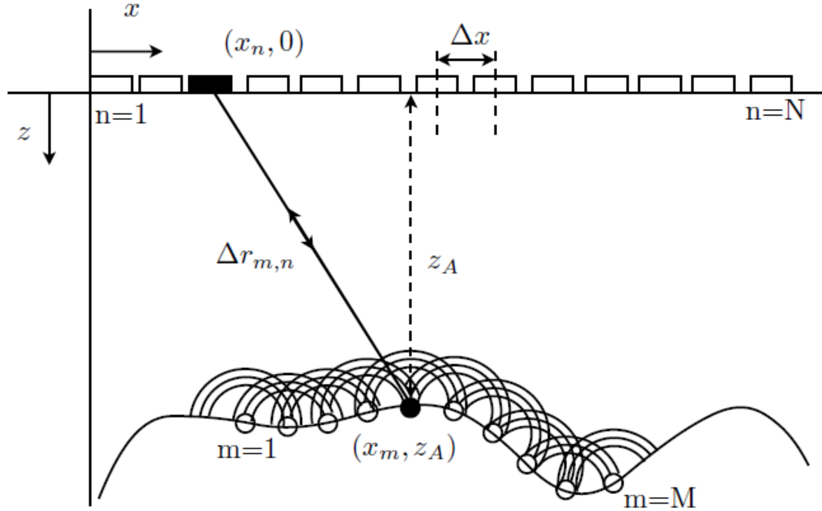


Figure 2.2: Defect in metal reflects ultra soon signals.

ber domain is, in this case, not an option because the transformation to the domain will lead to physical limitations[15] and is therefore excluded from this thesis project. The other two methods are discussed in this chapter. Table 2.1 the explains the parameters needed by the IWEX algorithm.

2.1 Time domain

First of all the algorithm defines a grid in the metal underneath the piëzo elements(see figure 2.3). The algorithm calculates the pressure for each element in that grid.

The listing 2.1 describes the main part of the IWEX algorithm(time domain). All listing in this document are Matlab scripts unless mentioned otherwise. For the com-

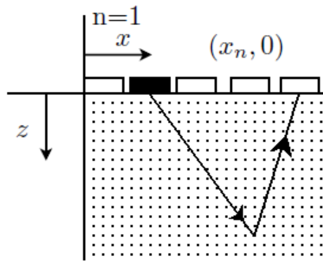


Figure 2.3: Grid defined in the metal.

plete script see appendix A. Appendix F gives a small explanation of common matlab expressions.

Listing 2.1: Time domain algorithm

```
1  Confidential
```

The acquired data is called 'in' and the output is called 'out'. The input data called 'in' contains all samples for every transmitter/receiver pair. The transmitter/receiver pair has a corresponding vector of samples which is stored in a column in 'in'. The 'in' matrix has a dimension of Number of samples (N_SAMPLES) in one direction and N_el^2 in the other direction. The output(out) contains the extrapolated pressures in the metal. As shown in the listing the algorithm consists of two loops. The first loop handles the transmitting elements and the second handles the receiving elements, both are thus an iteration from 1 to the number of piëzo elements. The second line calculates the x distance from the current sending element to the reference point which is the first element. x_A is a distance matrix that consists, for each matrix element, of the distance from the grid point to the reference point (X dimension). z_A is the same but in the Z dimension. The resolution of the final picture is defined by these matrices. x_A and z_A are used in line three which calculates the distance of the current transmitting element to all points in the matrix elements. Line four calculates the Geometric Spreading (GS) matrix. At line six a subset is created from the acquired data samples, this line takes only the data that is received by piëzo element nri when piëzo element nti transmitted(results in a vector). Line 7 and 8 calculate the distance from the receiver element to the grid point. The next line (9) adds these distances and divides them with the velocity of the signal through the metal. This generates a matrix which consists of times. Each element contains a time needed by a signal to travel from a particular transmitter (nti) to receiver (nri) element via a grid point. Line 10 calculates the needed samples based on the sample frequency and line 11 makes sure that the needed samples cannot exceed the maximum number (number of samples). t_i consists of elements containing a sample number that corresponds to that particular point in the metal. If there is a defect in the metal, it is likely that that particular sample will have a increased amplitude. Line 12 takes the indicated samples and multiplies them GS matrix. All the immediate results of all transmitting and receiving elements are added in line 13. When the algorithm ends, the 'out' matrix will have the final result containing the pressures in the metal at the defined grid points. As described earlier line 10 rounds the traveled distance based

on the sample frequency to the nearest natural number. This quantization lowers the resolution of the algorithm and therefore the time domain algorithm is undermined to the frequency domain with respect to the result.

In figure 2.4 an example picture of the IWEX algorithm using the time domain is illustrated.

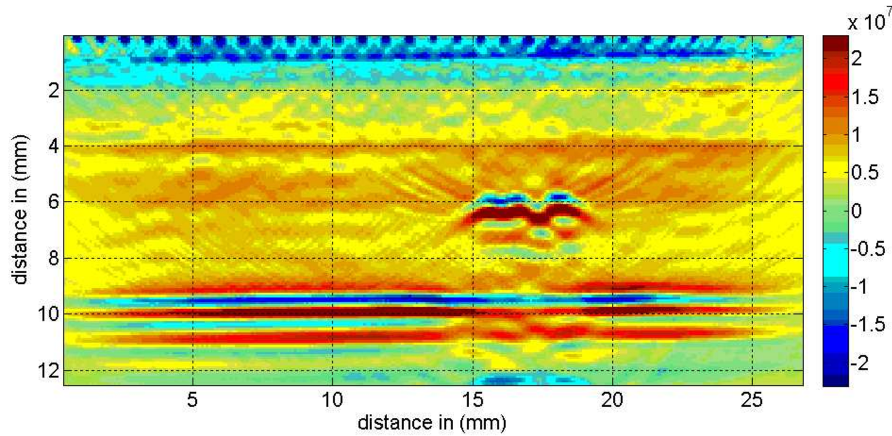


Figure 2.4: Example IWEX picture of time domain

2.2 Frequency domain

Another version of the algorithm is to convert the method to a calculation in the frequency domain. The conversion was done by Niels Pörtzgen and is listed in listing 2.2. For complete script see appendix D.

Listing 2.2: Frequency domain algorithm

```
1  Confidential
```

The basic technique is the same, but there are fundamental changes. First of all instead of iterating over the transmitter and receiver elements, the frequency domain iterates over the depth (z direction) and the frequency f. In line 1 the fast Fourier transformation is calculated of the acquired data. According to [15] the algorithm needs only the first couple of frequencies. These are indicated by i_f_start and i_f_end. Therefore the input data can be reduced which is done in line 2. The main calculations are done with two matrix multiplications on line 15.

In figure 2.5 an example picture of the IWEX algorithm using the frequency domain is illustrated.

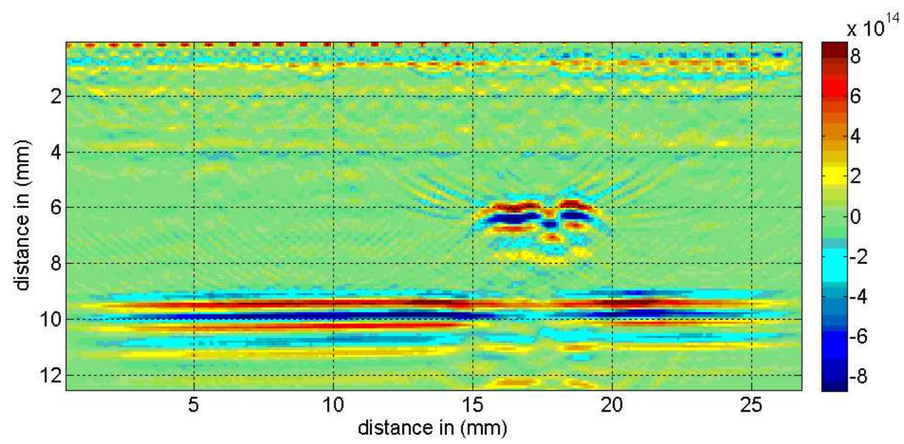


Figure 2.5: Example IWEX picture of frequency domain

Parameters and complexity of the algorithm

3

The computation and memory complexity of the algorithm depends on different parameters that are specified for the problem. Therefore we will first describe these parameters and next explain how they influence the computation and memory complexity of the algorithm.

3.1 Parameters

In table 3.1 all the parameters for the algorithm are specified, these parameters can be controlled for example by an operator.

Name	Min	Max	Dimension	typical
d_element	0.4	2	mm	0.85
d_wall	5	30	mm	30
cl_1	5900	5900	m/s	5900
N_el	32	128		128
i_f_start	1	1		1
i_f_end	150	150		150
xi_ROI_start	1	128		1
xi_ROI_end	1	128		128
zi_res	1/10	1/5		1/5
xi_res	1/10	1/5		1/5
Number of samples				1600

Table 3.1: Variables

In this thesis we have made some generalization. These are described as:

$$X_dim = \frac{xi_ROI_end - xi_ROI_start}{xi_res} + 1 = 636 \tag{3.1}$$

$$Z_dim = \frac{zi_res \times d_element \times round(1.25 \times \frac{d_wall}{zi_res \times d_element}) - zi_res \times d_element}{zi_res \times d_element} + 1 = 221 \tag{3.2}$$

$$F_dim = i_f_end - i_f_start + 1 = 150 \tag{3.3}$$

3.2 Time domain

First of all the acquired data has a size of $N_el \times N_el \times number_of_samples \times size_of_datatype$. The size of the datatype is determined by the hardware and is currently 12 bit. This gives a total of 39.3 MegaByte (MB)($128 \times 128 \times 1600 \times 12/8$)of

data that is needed for one iteration of the algorithm. Now lets take a look at the computational requirements. Table 3.2 calculates for each line in listing 2.1 the number of computations.

Line number	Calculation needed	
2	$N_el \times 1$	128
3	$N_el \times X_dim \times Z_dim \times 5$	90.0 M
4	$N_el \times X_dim \times Z_dim \times 1$	1.8 M
6	$N_el \times N_el \times 3$	49 k
7	$N_el \times N_el \times 1$	16 k
8	$N_el \times N_el \times X_dim \times Z_dim \times 5$	11.5 G
9	$N_el \times N_el \times X_dim \times Z_dim \times 2$	4.6 G
10	$N_el \times N_el \times X_dim \times Z_dim \times 2$	4.6 G
11	$N_el \times N_el \times X_dim \times Z_dim \times 2$	4.6 G
12	$N_el \times N_el \times X_dim \times Z_dim \times 1$	2.3 G
13	$N_el \times N_el \times X_dim \times Z_dim \times 1$	2.3 G
Total		30.0 G

Table 3.2: Computational requirements of the time domain

Note that all operations are counted as one, so multiplication, dividing, square root and squaring are all equal computational wise. The required number of computations is in the order of $O(N_el^2)$ which gives a total of 30.0 Giga(10^9) FLoating point Operations Per Second (GFLOPS).

3.3 Frequency domain

The acquired data has of course the same size, only the data that is used by the algorithm is fast Fourier transformed and only a subset of frequencies are used. Therefore the data set used by the algorithm has a size of $N_el \times N_el \times F_dim \times datatype_size \times 2$. The factor 2 is a result of the FFT which results in a data set of complex numbers. Claudiu Zissulescu[23] has researched the datatype_size and came to the conclusion that 8 bit FFT numbers can be used with an acceptable lose in precision. From this conclusion the data used by the algorithm will be 4.9 MB ($128 \times 128 \times 150 \times 1 \times 2$). Again lets take a look at the computational requirements. Table 3.3 calculates for each line in listing 2.2 the number of computations. The FFT complexity is dependent on the implementation and we have assumed that N is a power of 2. The required calculations for the algorithm are in the order of $O(N_el^2)$ which gives a total of 5.6T FLoating point Operations Per Second (FLOPS).

3.4 Number of samples

The memory requirements are dependent on the number of samples needed to represent the original signal and therefore we want to minimize the number of samples. The theoretical maximum distance can be calculated and thereby the maximum samples

Line Number	Calculation needed	
1	$N_el \times N_el \times (4 \times N \times log_2 N - 6N + 8)$	570.1 M
5	$Z_dim \times (X_dim \times N_el \times 3 + 1)$	54.0 M
8	$Z_dim \times F_dim \times 3$	99.4 k
9	$Z_dim \times F_dim \times 1$	33.1 k
11	$Z_dim \times F_dim \times (2 + X_dim \times N_el + 3 + X_dim \times N_el \times 7)$	21.8 G
16	$Z_dim \times F_dim \times N_el \times N_el \times X_dim \times 8 \times 2$	5.5 T
20	$Z_dim \times (F_dim \times X_dim + 3)$	21.1 M
Total		5.6 T

Table 3.3: Computational requirements of the frequency domain

needed for the algorithm. The maximum distance traveled is reached when the first element sends, the signal travels to the maximum wall thickness and the maximum xi_ROI_end and is reflected to the first element. In that situation we will need 3875 samples for 128 piëzo elements(see equation 3.4.

$$max_samples_number = 2 \times \frac{\sqrt{((xi_ROI_end - 1) \times d_element)^2 + (zi_res \times d_element \times round(\frac{1.25 \times d_wall}{zi_res \times d_element}))^2}}{cl_1 \times dt} \tag{3.4}$$

Note that the algorithm will never use all samples if the Region Of Interests (ROI) is smaller than all elements. In that case some samples could be dropped. If we have a ROI in the middle the first samples from the first element will never be used. If we use the ROI of 60 in the center we would need 2827 elements on average.

These calculations are correct if we assume that the piëzo elements can receive the signals in all directions. In a practical situation this is not true. [15] showed that if the pitch (d_element) between the elements is 0.85 mm the receiving angle is about 45 degrees. This would mean that the maximum distance in the x-axis is reduced. The calculations become:

$$max_sample_number = 2 \times \frac{(zi_res \times d_element \times round(\frac{1.25 \times d_wall}{zi_res \times d_element}))}{cos(45) \times cl_1 \times dt} \tag{3.5}$$

If we calculate 3.5 for 128 elements the maximum number of samples is 1801.

3.5 Single precision vs double precision floating point

The original algorithm was tested in Matlab. Matlab default uses double precision floating point for its calculations. However a different platforms get the most performance in single precision algorithm. But also the required memory and bandwidth is effected with this decision. This section compares Single Precision (SP) and Double Precision (DP) floating point calculations from a practical point of view.

For the comparison we have used the algorithm listed in 2.2. The algorithm is executed in Matlab two times. One time with double precision and the second time with single precision. The results of both executions are compared to each other.

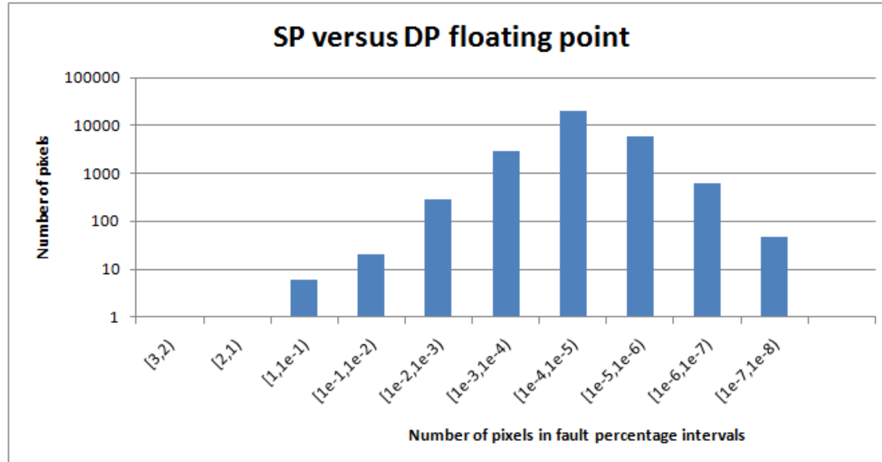


Figure 3.1: Number of pixels divided in fault percentages intervals

Figure 3.1 illustrates the number of pixels divided in fault percentages intervals. The x-axis consists of the intervals and the y-axis illustrates the number of pixels in that interval. Notice that the y-axis has a logarithmic scale. As can be seen in the picture most pixels have a fault percentage of less than 1e-4 %, and all pixels have a fault percentage less then 3%. Concluding that single precision floating point operations can be an option if necessary. Appendix H contains a second method of comparing single precision and double precision floating point computations.

3.6 Conclusions

If we compare the computational requirements for the time domain with the frequency domain we see that the frequency domain needs a higher order of computations(30 GFLOPS versus 5.5T FLOPS). This is mainly the result of the iteration over all frequencies (F_dim).

It is possible for the frequency domain to split up the FFT calculations and the algorithm itself. The number of samples for both algorithms is the same but if we split up the algorithm and FFT in the frequency domain, then the algorithm itself needs less data because it only needs the first 150 frequencies. In numbers this method reduces from 39.3 MB needed in the time domain to 4.9 MB used by the frequency domain for each frame.

It is possible to use single precision floating point arithmetic for calculating the IWEX algorithm.

4 Platform exploration

As seen in the previous chapter the IWEX algorithm needs a lot of calculation power, therefore we are looking at platforms which are able to supply that computational power. For this thesis we evaluated as candidates the Cell processor, FPGA and GPUs. In the next sections we will discuss each platform as well as how the IWEX algorithm can be implemented on that platform. Besides the selected platforms there are some other interesting platforms. Tilera Corporation developed the Tile64 processor which consists of a mesh of 8x8 VLIW processors. The processor uses 23 W for 80 GFLOPS, which is a high performance per watt[18]. Clearspeeds CSX700 processor exceeds that with 12 W of power for 96 GFLOPS[7]. Intel is working on a combined CPU and GPU called the Larrabee. The Larrabee architecture consists of many x86 cores which are based on the old Pentium architecture. The estimated performance of the Larrabee is 2 TFLOPS in single precision. The first chip with Larrabee architecture is expected in the first quarter of 2010[17].

4.1 GPU

Graphics Processing Unit (GPU) are originally designed for the game industry. The GPUs where originally not programmable (totally dedicated), only the last couple of years GPUs have become more programmable and, with more than 200 stream processors, more interesting for general purpose programming. Currently there are two major companies that design GPUs that can be used for general purpose: Nvidia and AMD/ATI. For this project we have chosen to use Nvidia, because currently the Nvidia development environment is more advanced than the AMD/ATI variant. In this section we give an overview over the latest Nvidia GPU architecture (GT200), and how this architecture can be used for the IWEX algorithm. The Nvidia Geforce GTX285 will be used as an example.

4.1.1 Architecture

A GPU is designed for graphic processing, this has resulted in a chip that is mainly dedicated for highly parallel computation and therefore has more area specified for floating point operations instead of control or cache (see figure 4.1)[14].

The architecture of a GPU is illustrated in figure 4.2. A GPU has multiple(N) multiprocessors. Each multiprocessor consists of more (scalar) processors(M) that share the instruction unit.

Each multiprocessor has a constant cache and a texture cache shared by all scalar processors in that multiprocessor. Note that the caches are relatively small compared to caches in a CPU. Each processor in the GPU is connected to the device memory. The

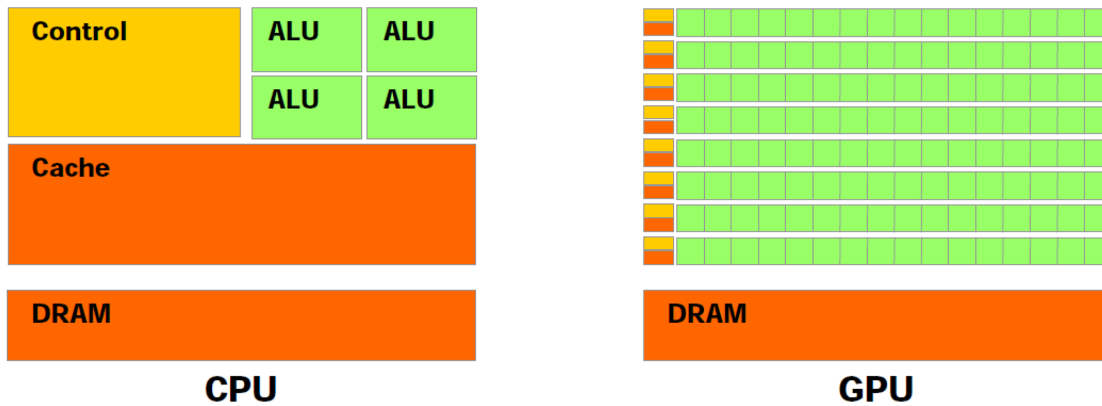


Figure 4.1: CPU vs GPU [14]

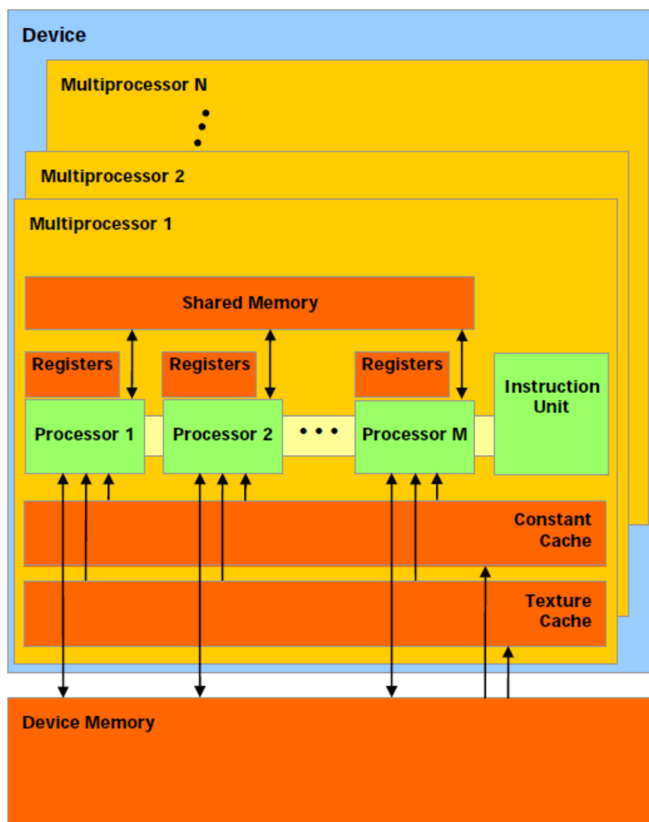


Figure 4.2: Architecture of a GPU [14]

Nvidia GT200 architecture has 30 multiprocessor (N) all consisting of 8 processors (M) and one double precision Arithmetic Logic Unit (ALU) so the GT200 has a total of 240 (30×8) (scalar)processors. Each processor can calculate 3 floating point operations per clock one multiply-add operation and one multiply operation. With a clock frequency

Scalar processors	8
Registers	16384 (32 bit)
Shared memory	16 KB(16 banks)
Constant cache	8 KB
Texture cache	6 to 8 KB

Table 4.1: Properties of a multiprocessor

of 1.296 GHz the GPU has 933 GFLOPS in single precision and 77 GFLOPS double precision ($1.296 \times 30 \times 2$). Table 4.1 shows different properties of the multi-processor.

The GTX285 GPU has 1 GB of Graphics Double Data Rate 3 (GDDR3) memory with a bandwidth of 159 GB/s. For communication with the CPU it uses a PCI Express 2.0 connection using 16 lanes and therefore having a theoretical bandwidth of 8 GB/s.

Nvidia has developed CUDA which is an extension to C for programming their GPUs. A function that has to run on a GPU is called a kernel. A kernel can be executed in multiple threads. Threads are grouped together and form a block and a block will be scheduled on a multiprocessor. The multiprocessor will schedule the threads over the different scalar processors.

The architecture of a GPU is called Single Instruction Multiple Thread (SIMT) by Nvidia[14] which is almost the same as Single Instruction Multiple Data (SIMD) but the key difference is in the notion to the programmer. With SIMD the width of the SIMD machine is exposed to the programmer. This is not the case with GPU. With GPU the warps and branches in the thread are at most important for high efficiency.

Each thread is mapped on a single scalar processor core. Threads are executed in groups of 32 called warps. Every clock cycle the SIMT unit selects a warp that is ready to execute(scoreboarding). A warp issues one common instruction each time. To get full performance all the threads in a warp should have the same execution path. Each GPU can have 32 active warps and 1024 active threads at a time per multiprocessor.

Although we are talking about threads, it is not entirely the same as with a CPU. A thread on a CPU has always all the resources from the CPU available. With a GPU thread this is not the case. A GPU has one big register file which has to be shared between all active threads. This is also the case for the shared memory.

4.1.2 Related research

[13] is a white paper from Nvidia which shows how to accelerate MATLAB with CUDA using MEX Files. They claim a speedup from 1.8 to 15.7 times.

[4] looked into GPU for calculating Cholesky factorization and used padding, hybrid GPU-Central Processing Unit (CPU) and recursion techniques to improve performance of the GPU. The result was a maximum of 159.5 GFLOPS on a device capable of 933 GFLOPS. The best performance on a AMD quad core was 59.6 GFLOPS so a speedup of 2.6 times based on the measured calculation performance.

[9] looked at matrix multiplication using the CUBLAS library from Nvidia, and determined that a GPU outperforms a CPU for about 78 times faster. But the rate really depends on the matrix size due to the slow host-gpu bandwidth.

4.1.3 Mapping of the algorithm

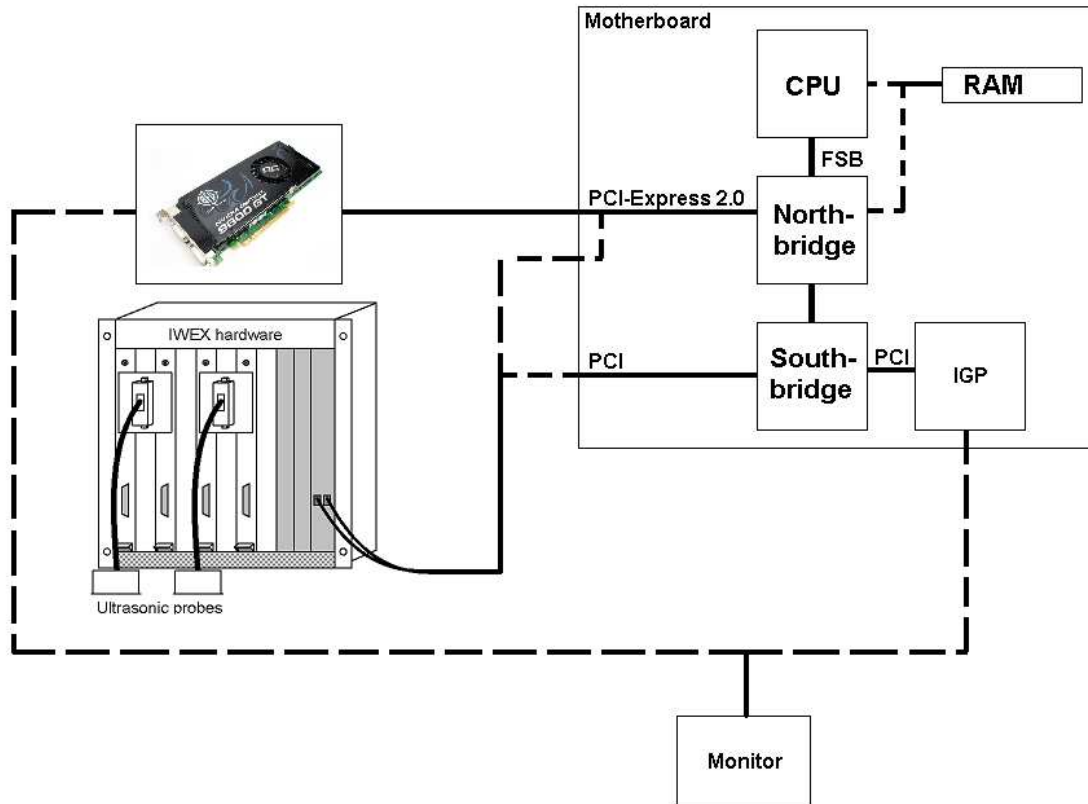


Figure 4.3: Hardware needed for the GPU solution

Figure 4.3 illustrates the hardware needed for a GPU environment. The continuous lines are links that are necessary, the others are options from which one should be chosen. At first there should be a motherboard with a CPU and memory. Dependent on the CPU, the memory is attached to the CPU itself or the north bridge. Intel's core i7 has the memory controller integrated on-chip, while all older versions had a north-bridge with a memory controller. The motherboards should support the Video card. The latest generation video cards support PCI-Express 2.0. And in our case the hardware needs a PCI extension card. The monitor can be attached to a integrated GPU on motherboard or it can be attached to the GPU.

Note that it it possible to add multiple GPUs to one motherboard. Nvidia currently support up to eight GPUs in a single machine. The GTX295 card has two GPUs per card, so for a total of eight we need four cards in a single PC using up four PCI-Express slots, preferably 16 lanes per slot. Therefore needing a total of 64 lanes. If the motherboard does not support 64 lanes the cards will run on eight lanes which reduces the bandwidth with a factor of two. One of the drawbacks of a GPU is the energy consumption, each dual GPU card uses 289 Watt.

Until now we have only spoken over the theoretical maximum FLOP count. But

this maximum performance is almost impossible to get for different reasons. One major reason is that the scalar processors are able to execute a multiply-add instruction and a multiply instruction in one clock cycle but the multiply instruction is not part of the ALU but is used for interpolation and perspective correction[16]. Most of the time this multiplier is not used in combination with CUDA. Looking at the ptx code, what resembles assembler code for the GPU, generated by the compiler the multiplier is not used and therefore the maximum of FLOPS is reduced to 622 ($240 \times 2 \times 1.296$) GFLOPS instead of 933 GFLOPS.

4.1.3.1 GPU Software time domain

There are two major options to calculate the time domain algorithm on the GPU. The first one consists of pre-calculating whatever you can and storing it in the global memory and then load the pre-calculated data from memory when needed in the critical part of the algorithm. The second option is to calculate the values when needed, without pre-calculating (original algorithm). The algorithm using pre-calculation is listed in 4.1.

Listing 4.1: Time domain algorithm with look-up tables

```
1  Confidential
```

The algorithm starts with creating space for the pre-calculations. The space should eventually reside in the GPUs global memory. The second part of the algorithm pre-calculates the GS and t_i_n matrices. Those matrices are used in the last part, this part is the critical part and should be repeated for every new set of data. If this algorithm is implemented on a GPU the thread code should do the following:

Listing 4.2: GPU kernel speudo code for time domain

```
1  Confidential
```

As is illustrated in the listing there are at least two reads necessary for one calculation. This means that the main bottleneck will be the memory latency. The used GPU has a bandwidth of 159 GB/s. If we assume 128 elements then we have 128 times 2 global load in the inner loop and one in the outer loop, given a total of 32896 loads for one pass of the algorithm resulting in a single pixel value (one matrix element). In that pass the algorithm has calculated 32768 FLOPS($128 \times 128 \times 2$). If we calculate the number of loads for the whole matrix we have 4.6G loads per pass ($X_dim \times Z_dim \times N_el(1 + N_el \times 2) = 636 \times 221 \times 128(1 + 128 * 2)$). If we assume single precision floating point operation, we have 18.5 GB per pass. This should take 116ms ($\frac{18.5}{159}$).
Besides the memory transactions there are also computational requirements. There are at least 513 computations ($N_el \times 4 + 1$) to calculate 256 Floating Point OPeration (FLOP)s ($2 \times N_el$), which is a efficiency of 49.9 % ($\frac{256}{513}$). If we calculate that for the complete GPU we get 310 GFLOPS (622×0.499). The number of FLOPS needed for one pass is 4.6 G ($2 \times N_el^2 \times X_dim \times Z_dim = 2 \times 128^2 \times 636 \times 221$). In case the calculation is not done while loading data, it takes 15 ms ($\frac{4.6}{310}$) to calculate the result.
The maximum duration of the algorithm with pre-calculation should be 15+116=131 ms. Note that in this case we assume the theoretical maximum bandwidth. For this

algorithm is it hard to achieve that theoretical performance simply because the address for the memory load is calculated with a variable that is dependent on the position of the transmitter and receiver. This will lead to non coalesced memory reads. Transportation between host and device memory also takes up time. In the critical path we need to transport the sensor data to the GPU memory and the result from the GPU. The sensor data matrix has a size of 104.8 MB. ($numberOfSamples \times N_el \times N_el \times sizeOfDatatype = 1600 \times 128 \times 128 \times 4$). The result matrix has a size of 0.56 MB. ($X_dim \times Z_dim \times sizeOfDatatype = 636 \times 221 \times 4$). These have to be transported through PCI-Express 2.0 16x, which is capable of 8GB/s in each direction. 104.8MB transporting over 8GB/s takes 13.1 ms. ($\frac{0.1048}{8}$). This time has to be added if you don't overlap transporting and calculations.

4.1.3.2 GPU Software frequency domain

CUDA has a Basic Linear Algebra Subprograms (BLAS) library and therefore the first simple solution would be to implement the algorithm on the CPU and offloading the matrix multiplications to the GPU using the library. This would generate a lot of data transfers to and from the GPU. Since that link has a relatively low bandwidth it may be advantageous to implement the complete algorithm on the GPU and therefore only have to send the acquisition data and get back the result. If we put all the matrices in the DRAM and make a simple implementation that all scalar processors get the data from DRAM, then for every multiply-add instruction we need two reads from memory(8 bytes) for two FLOPs. Bandwidth of memory is 159 GB/s so this mapping has a memory limit of 40 GFLOPS($\frac{159}{4}$), while the GPU has 933 GFLOPS. This means a source utilization of 4.3 % and so this mapping is not the best solution.
Another solution for this problem is to reuse data obtained from the memory. This can be done using the shared memory. Each multiprocessor has 16 KB of shared memory. A method to reuse data is to split the matrices in sub-matrices and do the calculations for the sub-matrices local in the shared memory. This method is explained in [12]. With this method each source sub-matrix block can be used multiple times. Each sub-matrix must map to a grid block, and create a thread grid that maps to all matrix elements in that block. Let every thread calculate the value for one particular matrix element of the result of the multiplication by using shared memory. It can be proved that the number of times that data of an element is reused is the same as the dimension of the sub-matrix. Of course this number is limited to the size of the shared memory. A single precision floating point data type is 4 bytes width. If we look at the bandwidth of 159 GB/s we can get 39.75 floating point numbers per second. So if we want to saturate the calculation performance we need to $\frac{933}{39.75} = 23.5$ FLOP per memory access. This is theoretically possible as we take a sub-matrix size of 32. Then we would create a grid of $32 \times 32 = 1024$ threads in one block, but the limit is 512, so we have to use a block size of 16. Note that we are multiplying two complex matrices. This means that we have to load twice as much data but an operation also takes multiple cycles. For example a multiply-add for a complex number consists of four multiplications and four additions.

The GPU does besides useful floating point operations also loading and storing of data, calculation of different indexes.

Listing 4.3: Frequency domain overhead

```
1  calculate left matrix index
2  calculate right matrix index
3  load left matrix element
4  load right matrix element
5  clear loop variable i=0;
6  clear temp_r and temp_i
7  loop:
8    calculate complex multiplication-addition temp_r+ = ac-bd; temp_i += ad+
9    increase loop variable
10 if i<block size goto loop
```

Listing 4.3 calculates 128 (16×8) useful FLOPS(assumed that matrices are in global memory with block size of 16). The overhead for doing those calculations are 119 instructions($7 + blocksize * 7$). In reality the calculation of the indexes takes more instructions, most likely dependent on the thread and grid positions in the grid, which gives four variables. This will lead to the form of ïndex = bx*constant+by*constant+tx*constant+ty*constant". This will take at least 7 instructions, so the overhead will increase to 133 (19+block size*7), so the efficiency reduces to 49 % ($\frac{128}{128+133}$). 49% of 622 GFLOPS is 305 GFLOPS. To decrease the overhead we can use loop unrolling, in that case the loop variable and all connected statements disappear and we are left with the overhead of just the index calculation and loading which is 82 instructions ($18 + blocksize * 4$) and a corresponding efficiency of 61 % ($\frac{128}{128+82}$). 61 % of 622 GFLOPS is 379 GFLOPS. Note that this performance only holds up when there are enough threads to overlap the memory latency. The frequency domain algorithm needs 5.5 TFLOPS. This means that one pass of the algorithm will take up $\frac{5.5}{0.379} = 14.5$ seconds.

4.1.4 Conclusions

- A GPU is capable of calculating the IWEX in both domains.
- The GTX285 has 933 GFLOPS theoretical but due to overhead in index calculation it reduces to 379 GFLOPS in the frequency domain and 310 GFLOPS in the time domain.
- The execution time for one image is expected to take 144ms (if nothing is overlapped) for the time domain and 14.5 seconds for the frequency domain.

4.2 FPGA

A Field Programmable Gate Array (FPGA) consists of gates (or logic blocks) that can be configured in field to perform dedicated functions. For the generation of the configuration a Hardware Description Language (HDL) is used. For the IWEX algorithm we made the distinction between fix-point calculation and floating point calculation because of the performance differences in both cases. In this section we try to estimate the performance in fix-point and floating point but also for the time domain and frequency domain.

Device	Number of slices	Number of DSP slices
XC5VLX330T	51,840	192
XC5VSX240T	37,440	1,056

Table 4.2: Devices used for estimation

Operation	DSP usage	Number of slices	Frequency (MHz)
Multiply	0	330	334
Multiply	1	161	379
Multiply	2	74	410
Multiply	3	53	410
Adder	0	295	359
Adder	0	246	435
Adder	2	145	410

Table 4.3: Operations and their resource usage

4.2.1 Floating point performance with respect to the frequency domain

In this section we try to estimate the performance of a FPGA with respect to the maximum number of FLOPS it can handle. To determine the maximum floating point performance we look at the maximum number of resources and the number of resources a floating point operation requires. Divide that with each other and in combination with the frequency of the FPGA one can calculate the maximum performance[1]. For this comparison we use only single precision floating point operations.

The first question is what kind of operations we need for the algorithm. As the IWEX frequency domain algorithm mainly consists of matrix operations, the operations are equally spread between multiplying and adding. The next problem is the interfacing with the floating point cores and the memory needed to supply the data to the cores.

As test platform we take two different Xilinx virtex 5 FPGAs. Table 4.2 shows some of the specifications of these devices.

We need floating point adders and multipliers. This can be implemented with pure logic or dedicated Digital Signal Processor (DSP) logic. Table 4.3 shows the different functions with their resource occupation [22].

The number of slices is calculated as the number of LookUp Table (LUT)s + number of Flip-Flop (FF)s and the total divided by four. Note that this depends on implementation choices.

We have two devices and four choices in multipliers and three choices of multipliers which gives a total of 24 distinct solutions. If we calculate the number of FLOPS for each solutions the quickest is the XC5VSX240T with max usage of the DSP slices (for adders and for the multipliers) with a maximum performance of 134 GFLOPS. For the

estimation we are using the following equations:

$$MFLOPS = \frac{Number_of_slices_available - 5000}{Number_of_slices_adder_and_multiplier} * \min(frequency_adder, frequency_multiplier) \quad (4.1)$$

$$MFLOPS = \min\left(\frac{Number_of_slices_available - 5000}{Number_of_slices_adder_and_multiplier}, \frac{Number_of_DSP_slices_available}{(Number_of_DSP_slices_adder_and_multiplier}\right) * \min(frequency_adder, frequency_multiplier) \quad (4.2)$$

Equation 4.1 is valid when no DSP blocks are needed. If they are needed we have to use equation 4.2.

As you can see in the equation we subtracted 5000 slices, these slices can be used for a communication interface. Also note that because of the min function in the equation there may be some space left on the device because of the restriction of the other resource.

Furthermore it is likely that the clock frequency drops a bit for larger designs, and some extra LUTs are needed for wiring and mapping.

4.2.2 Fix-point calculation

Claudiu Zissulescu has researched the possibility for fix-point calculations for the IWEX algorithm in [23]. The conclusion was that is is possible, but dependent on the widths of the variables and the data path. If we take 8 bits for P_data and 15 bits for W_A_r the accumulators for the matrix multiplication should be 25 bit. For the summation over the diagonal 15 bits is required according to page 11 in [23].

The virtex 5 FPGA has so called DSP48E slices. These slices contain a 25x18 multiplier and a 48 bit accumulator (see [2]). A matrix multiplication can be handled with a so called multiplier - accumulator operation. Each operation (multiply or add) can be done by a DSP slice, but then every DSP block should be able to handle the bit widths. Each DSP slice runs on 550MHz. The Xilinx FPGA with the most DSP slices has 1056 slices, 1056 times 550 is 580 GOPS only with DSP slices. We could implement adders in logic but with the penalty of lowering the clock frequency.

4.2.3 Mapping of the algorithm

4.2.3.1 Time domain algorithm

Figure 4.4 illustrates a possible implementation of the time domain. The implementation is based on the idea that every FPGA serves a number of piëzo elements (for example 32) and calculates the final image for those elements. This implementation is based on pre-calculating the t_i matrices for each iteration, here called t_i_n. We start on the top. From the elements via the ADC the FPGA gets 1600 samples. These samples are stored in RAM (local memory). With the t_i the lookup block will take the correct samples and supplies it to the multiplier. The other input for the multiplier will be the geometric spreading matrix. The result of the multiplication will be added to the final result. If all data is handled the sub-result is completed and has to be added to the other three (for a total of 128 elements) to create the final image for displaying. The main problem lies in the bandwidth of the memories. For the total local memory used for storing the samples needs $32 \times 1600 \times 8 \approx 410k$bits. This can be implemented using on-chip RAM. The GS

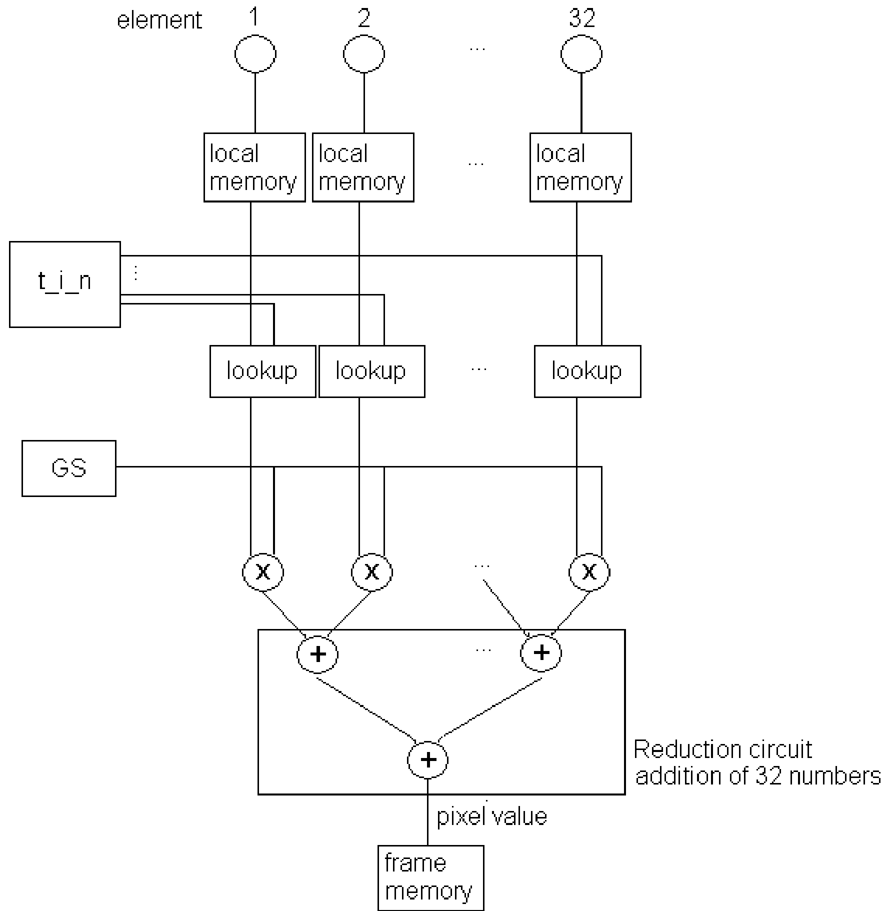


Figure 4.4: FPGA block diagram time domain

memory needs to be $128 \times 636 \times 221 \times 4 \approx 72M$ byte. This should be implemented in external memory and needs to be able to supply one element each clock cycle. The t_i_n matrix is more difficult. It has to produce $32 \times 2 = 64$ bytes of data per multiply cycle. The t_i_n requires a size of $636 \times 221 \times 128 \times 32 \times 2 \approx 1.1G$ byte. This is large so an option is to calculate it on the fly. Then we need to calculate the following:

$$x_t = (nti - 1)d_element \tag{4.3}$$

$$dr_t = \sqrt{(x_A - x_t)^2 + z_A^2} \tag{4.4}$$

$$x_r = (nri - 1)d_element \tag{4.5}$$

$$dr_r = \sqrt{x_A - x_r)^2 + z_A^2} \tag{4.6}$$

$$t_i = \lfloor \frac{1}{cl_1} dr_t + \frac{1}{cl_1} dr_r \rfloor \tag{4.7}$$

Figure 4.5 illustrates the algorithm to calculate one t_i element, therefore we need one of them for each piëzo element.

Notice that the calculation need to be pipelined for optimum throughput.

Confidential

Figure 4.5: Block diagram for calculating ti

As in the illustrated figure we need 13 extra arithmetic units. The dt_t needs to be calculated once for each pixel and can be distributed to every element. In total we need a total of 261 arithmetic units to serve 32 piëzo elements. Additional arithmetic units are needed to calculate the input variables. If we use fix-point calculation than each arithmetic unit can be implemented with one DSP unit and therefore can be implemented in one FPGA. If one FPGA is used to serve all piëzo elements (128) we need 1029 DSP units, which is also possible. If we want to use floating point arithmetic units we would need 684 DSP blocks to server 32 piëzo elements.

If we determine the performance of the algorithm we have to look at the number of cycles that is needed for every iteration over the transmitter. For every iteration the local memories are filled with samples and the FPGA should handle those samples before the next iteration. In that time it has to handle all receivers for all pixels. If we assume that each pixel/receiver element can be handled in one clock we need a total of 140556 cycles (636×221). If we look at fix-point calculation we can have a maximum clock frequency of 550 MHz. With 550 MHz one iteration takes 256ns. The number of iterations is equal to the number of piëzo elements and therefore one image takes 33 ms (128*256ns). If we look at floating point performance the clock frequency is bounded to 410 MHz and therefore it takes 44ms for one image. Notice that the FPGA does not have enough DSP blocks to server all piëzo elements at once, but to prevent the use of extra FPGAs it is possible to use multiplexers to switch between the sets of elements resulting in a execution time that is four times higher which also holds for the local memory.

4.2.3.2 Frequency domain

For the frequency domain the mapping is more difficult. Floating point operations are not comparable in performance with other platforms. Fix-point has the potential to be faster than other platforms. Fix-point has the disadvantage to be less accurate depending on the bit size. To get the most performance with a FPGA the data type must be less or equal to the size a DSP block can handle. The DSP block of a virtex 5 Xilinx FPGA can handle 25x18 bit multiplication and 48 bit accumulation. If we assume that P_data is 8 bit width and W_A_r element 16 bit[23]. Then the first matrix multiplication creates a bit width of 16+8+7=31 bits (matrix size of 601x128*128x128). The second multiplication leads to 54 bits (31+16+7) theoretical. The last operation is adding over all frequencies (150 times) which leads to 62 bits (54+8). For maximum performance the accumulation may not be larger than 48 bits. Claudiu showed in [23] that the bit sizes can be lowered with little error in the overall result. Therefore we assume that each operation can be performed with one DSP block. The algorithm mostly consists of matrix multiplications. If we use the FPGA to do a matrix multiplication then the FPGA does not have enough DSP blocks to do the calculation in one clock, so it has to be split up. We use the same algorithm as before and use sub-matrices. The FPGA is able to calculate a matrix of 11x11 ($\lfloor\sqrt{\frac{1056}{8}}\rfloor$) elements (complex) per clock cycle (fix-point). Supplying the DSP elements becomes thereby a problem. We have to load the two matrices that has to be copied in memory. W_A_r matrix has the dimensions of 636x128 each complex element consisting of 2 bytes is 325 kB ($636 \times 128 \times 2 \times 2$). P_data has a dimension of 128x128 each complex element consisting of 2 bytes is $128 \times 128 \times 2 \times 2 \approx 66$kB. With a total size of 374 kB this could fit a the memory of an FPGA. To supply the matrix calculation we need at least $2 \times 11 \times 2 = 44$ elements of 16 bit is 88 bytes per clock cycle. 88 times 550 MHz is 48.4 GB/s required for the memory.

For calculating 11x11 final matrix elements we need 128 clock cycles. For the x dimension we need $\lceil\frac{128}{11}\rceil = 12$ sub-matrices, for the y dimension we need $\lceil\frac{601}{11}\rceil = 58$ sub-matrices. In total we need $12 \times 58 \times 128 = 89088$ clock cycles. In this time it is also necessary to get the new data for the next matrix multiplication.

The time for a complete picture would be $89088 \times 221 \times 150 \times \frac{1}{550M} = 5.4$ seconds.

A second method for the implementation of the frequency algorithm is doing the calculation at the level of each sensor.

Figure 4.6 shows the block diagram for this approach. First the sensors generate the data, that data is Fourier transformed and stored in local memory. The local memory needs to store the values for all the needed frequencies for all firings ($150 \times 128 \times 2 = 38.4kB$). The next part takes one row from the W_A_r matrix and multiplies that with all the columns of the original P_data and stores the result in the second local memory (for one frequency). In the second iteration the multipliers do the same but for the next frequency, until all frequencies are handled. After that the next iteration will be the second row until all rows are handled for all frequencies. And if this is done it starts over with the second row in the Z dimension. So the sequence is for one row matrix multiplication over first all frequencies then all rows in W_A_r then in dimension Z. The third part of the algorithm takes the results out of the second memory and multiplies them to W_A_r transpose. And the result of that are again added to each

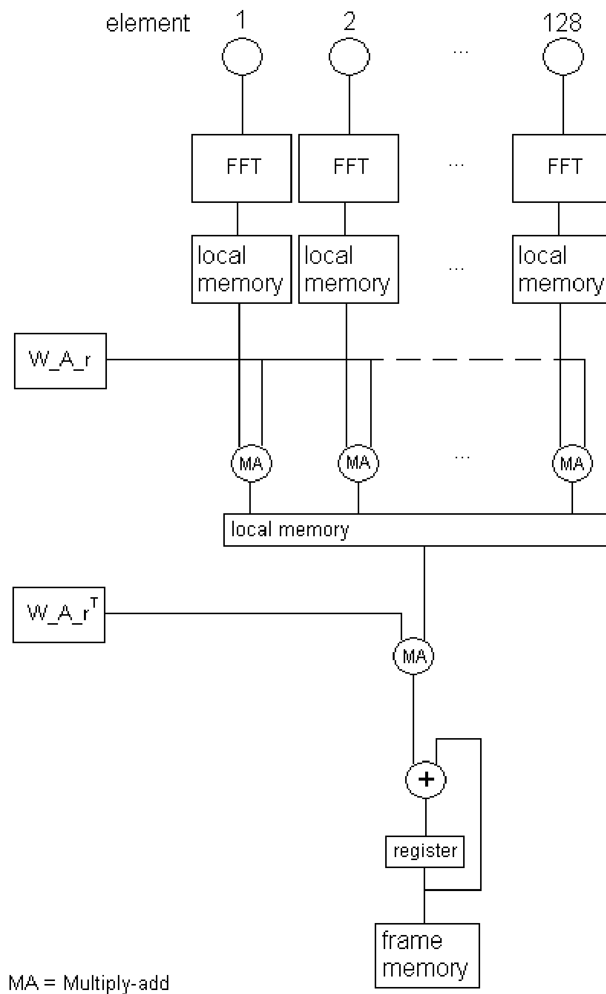


Figure 4.6: Block diagram for calculating in frequency domain

other and finally stored in the frame memory. This only works if you first iterate over the frequencies instead of the rows of W_A_r. So the second memory has to store all data that is generated in 128 cycles. That means that after two times 128 the value of the third stage is valid, and so after 19328 ($150 \times 128 + 128$) cycles one pixel value is calculated. So a complete picture would take 2.7 G cycles ($19328 \times 636 \times 221G$). Based on a clock frequency of 550 MHz this would take approximally 4.9 seconds for fix-point calculation. For this solution we need 1034 DSP blocks $(128 + 1) \times 8 + 2)$. Note that one complex Multiply-Add (MA) instruction takes eight DSP blocks. For the first local memory we need 4.9 MB ($128 \times 38.4k$). W_A_r needs a size of 20GB ($441 \times 150 \times 601 \times 128 \times 2 \times 2$). The size of W_A_r can be reduced if we use the toeplitz property of the matrix[15]. The W_A_r memories should be able to load two matrix elements in a clock cycle, therefore the architecture needs 4.4 GB/s ($2 \times 550 \times 2 \times 2$).

4.2.3.3 Remarks

In this section we have assumed that we can get the maximum clock frequency of the FPGA, but in practice this can be hard to realize. The wire delay as well as the fan out is a problem in current FPGAs, and therefore it may not be possible to design a system on the FPGA with the maximum frequency of 550 MHz.

4.2.4 Conclusions

- Computational power on a FPGA is not comparable with other architectures in this thesis if we look at floating point computations.
- An FPGA is an option if we use fixed point calculation.
- The computational load can be distributed over multiple FPGA almost in an ideal manner.
- The time domain can be fully implemented on one FPGA resulting in 33ms of calculation time for one image in fix-point calculation.
- The time domain can be implemented with floating point calculation resulting in 176 ms of calculation time for one image.
- The frequency domain can be implemented using fix-point calculation.
- The quickest described implementation of the frequency domain takes 4.9 seconds.
- For the GS matrix we need 1 GB/s of bandwidth for the external memory. For the W_A_r matrices we need 2 GB/s.

4.3 Cell

The Cell processor is a product that came from the cooperation of Sony, IBM and Toshiba. The main target was the Sony Playstation 3 console.

4.3.1 Architecture

In figure 4.7 the organization of the Cell processor is illustrated. The processor consists of eight so called Synergistic Processing Element (SPE) and one Power Processing Element (PPE), which has a PowerPC architecture 64 bit and 2 way simultaneous Multi-Threading. It has 32KB L1 cache and 512 KB unified L2 cache. The PPE and SPEs are connected with an Element Interconnect Bus(EIB) with a bandwidth of 204.8 GB/s. The EIB consists of 4 rings each 16 bytes width and full duplex. They are grouped in two with the difference that both groups runs in the opposite direction. The bus runs on the half of the core frequency which is 1.6 GHz. Which makes a total of 204.8 GB/s ($1.6G \times 16 \times 4 \times 2$(full duplex)). This is of course the limit and is dependent on which devices are communicating. The main memory bandwidth is 25.6 GB/s.

The PPE is a PowerPC that is responsible for coordinating the SPEs and running the operating system. The organization of the SPE is illustrated in figure 4.8. The SPE

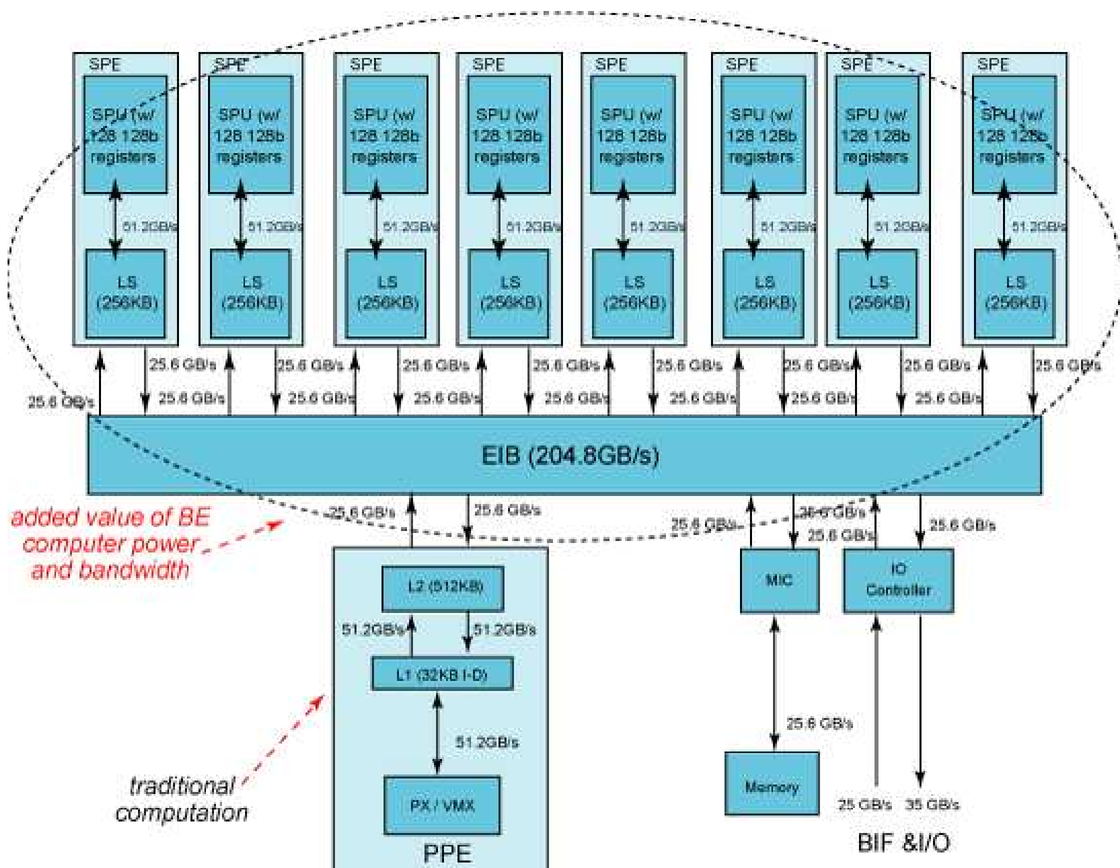


Figure 4.7: Architecture of a Cell [5]

consists of two execution units. One for floating and fix point calculations and one for the other instructions. It has a 256 KB of local memory and 128 128bit registers. The SPE has also a memory flow controller which is able to do Direct Memory Access (DMA) transfers and translating addresses independent of the execution units. The pipelines vary from two to seven cycles. The instruction fetch logic reads 32 instructions at a time into the buffer and two of them are fed at a time into the execution logic.

Each SPE can do four floating point operations per cycle and with a running frequency of 3.2 GHz they have a theoretic limit of 25.6($3.2 \times 4 \times 2$ multiply-add)).

The cell processors are currently available in three different systems. The first is the IBM BladeCenter QS22 computer with the cell as core processor. IBM uses this for super computers and put multiple blades in a chassis(6.4 TFLOPS in a BladeCenter H chassis). The cell processor is also available on a PCI-Express card, which can be used as a processor expansion board. The board has one 1 Gb ethernet port which can be used to write directly into the on board memory. And finally the Cell processor is used in the playstation 3 game console.

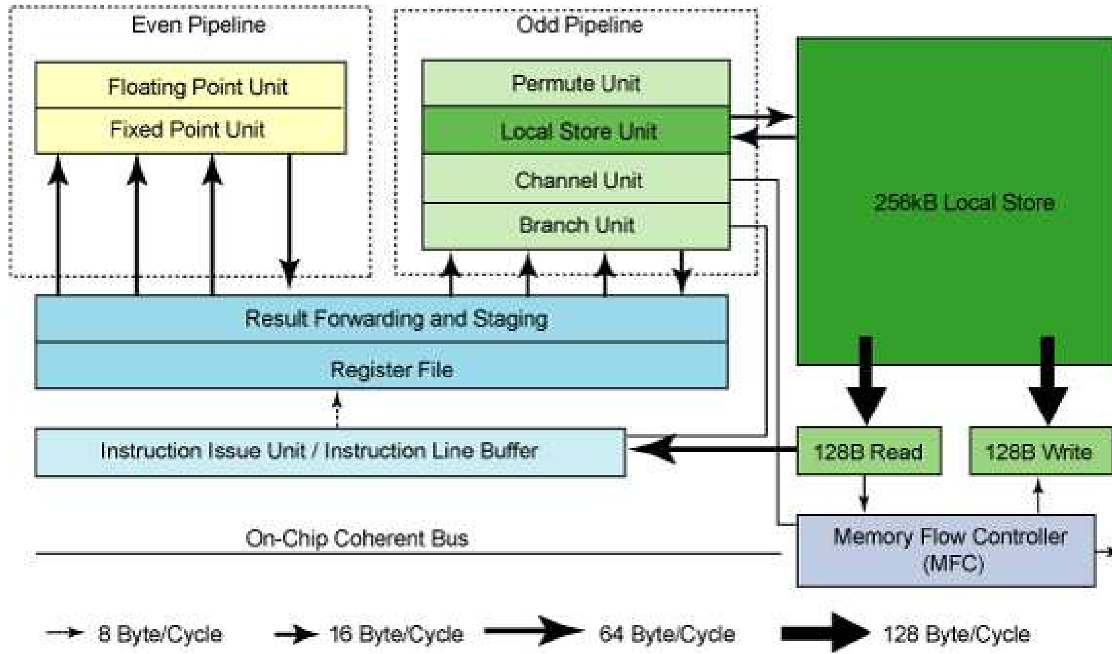


Figure 4.8: Architecture of a SPE [5]

4.3.2 Mapping of the algorithm

4.3.2.1 Frequency domain

For the frequency algorithm it is possible to pre-calculate the W_A_r matrices for every iteration. By doing that the processing unit only have to multiply matrices in the critical path. This principle is used to estimate the performance of the Cell processor. The Cell processor has a interconnection(Element Interconnect Bus (EIB)) with a high bandwidth, but the bandwidth of the memory is still limited to 25.6 GB/s therefore we need to hide memory latencies. We could use an algorithm that divides the matrix in sub-matrices. Since the main part of the frequency algorithm consists of a matrix multiplication, that multiplication can be done using sub-matrices. Due to the local memory size of 256k we could take a maximum dimension of the sub-matrix of 104 ($\lfloor\sqrt{\frac{256k}{3\times2\times4}}\rfloor$) in both directions based on storing three sub-matrices. The matrix dimensions are 636x128 (W_A_r) multiplied with 128x128 (P_temp) and therefor we take a sub-matrix dimension of 64. Every SPE can handle part of the Z dimension(outer loop) which gives an equal load for each SPE. A sub-matrix with the dimensions of 64x64 needs 32kB of memory ($128 \times 128 \times 2 \times 4$). The processor needs 1.3us ($\frac{32k}{25.6G}$) for loading a sub-matrix from main memory to local memory. For one sub-matrix multiplication we need to load two sub-matrices from main memory and the SPE calculates 2.1MFLOPs ($64^3 * 8$). Loading of the two sub-matrices takes 2.6 us and calculation of 2.1MFLOPs takes 82 us ($\frac{2.1M}{25.6G}$). So each SPE will take 2.6 us from main memory every 82 us and therefore the main memory is not the bottleneck even when there are eight SPEs.

By using this algorithm the system needs 20 GB ($221 \times 150 \times 636 \times 128 \times 2 \times 4$) for

storing the W_A_r matrices for each iteration. This can be reduced using the toeplitz structure of the matrix, and by using that only one row of the matrix has to be stored for each iteration. Reducing the memory size requirements to 161 MB ($221 \times 150 \times 636 \times 2 \times 4$). But this does not only effect the size of the main memory but also the bandwidth requirements because instead of loading sub-matrices of W_A_r it is now possible to load the one column for that iteration and reconstruct the needed sub-matrix from that column. For each iteration of the inner loop the SPE has to load four sub-matrices for the P_temp matrix and one column for the W_A_r matrix. This gives a total of 133kB ($64 \times 64 \times 2 \times 4 \times 4 + 636 \times 2 \times 4$) which will require 5.3 us. The number of calculations for one iteration is 83.6 MFLOPs which requires 3.3 ms. So each SPE will need 5.3 us main memory time every 3.3 ms.

Using this method we will be able to get the maximum FLOP rate of $25.6 \times 8 = 204.8$ GFLOPS. With the PPE available for control and the FFT. The maximum FLOP rate is possible because every SPE consists a floating point pipeline and a pipeline for all other instructions. The total execution time for one image is 26.8s ($\frac{5.5}{0.2048}$).

4.3.2.2 Time domain

For the time domain algorithm the division between the SPEs can be made by dividing the outer loop. Instead of iterating over all N_el each SPE has to do $\frac{1}{8}$ of all elements. In this way each SPE has the same load. Each SPE implements the whole algorithm and load for each inner loop iteration the data vector in shared memory. The vector has a size of 6.3kB (1600×4) and takes 250 ns to load from main memory. Using that data the algorithm calculates 2 MFLOPs ($\frac{32.3G}{128^2}$) which takes 78 us. Therefore the memory bandwidth will not be the bottleneck. The total execution time for one image is 158ms ($\frac{32.3}{204.8}$).

4.3.2.3 Data transport

The data that is required by the piëzo elements must be transported to the main memory of the Cell processor. currently the faster method is to use PCI-Express 2.0 with 16 lanes which has a theoretical bandwidth of 8 GB/s. The data is a matrix which in one dimension is equal to the number of samples and in the other dimension dependent of the number of piëzo elements. Using the indicated typical values the total size is 52.5 MB ($1600 \times 128 \times 128 \times 2$) based on 16 bit samples. Therefore for each image we have to transport that size which will take 6.5 ms. In the worst case scenario the transportation cannot be overlapped with calculation and this time has to be added for the total time. Result then will be 165 ms needed for one image.

4.3.3 Conclusions

- The Cell processor has two pipelines and therefore it is easier to acquire maximum performance.
- The execution time of the frequency domain is estimated on 26.8 s.
- The execution time of the time domain is estimated on 165 ms.

Platform	Theoretical GFLOPS	Memory bandwidth GB/s	Estimated execution time	
			Time domain ms	Frequency domain s
GPU	933	159	144	14.5
FPGA	580[1]		33/176[2]	4.9
Cell	204.8	25.6	165	26.8

Table 4.4: Overview of the properties of the different platforms

4.4 Conclusions

Table 4.4 gives an overview of the described platforms and its properties regarding the IWEX algorithm. If we look at the time domain the FPGA is able to calculate one image the fastest in fix-point calculation, but it is the slowest if floating point calculation is used. The GPU is the fastest in floating point calculation. The frequency domain has the same property, the FPGA is the quickest using fix-point calculation then the GPU and the Cell processor is the slowest.
So the conclusion of chapter is that an FPGA is the quickest solution in fix-point computation and the GPU in the case of floating point computation. An FPGA is a lower level architecture, and therefore can be suited to your needs but it is also more difficult to acquire the computational requirements. Notice that for the implementation on the FPGA we would need a big and expensive FPGA.

Because Chess has a lot of experience with FPGAs, but not with GPUs, we have decided to implement the IWEX algorithm on a GPU.

Implementation 5

5.1 Implementation time domain

In this section we discuss the mapping of the time domain algorithm on the GPU. The chapter begins with the matlab implementation and rewrites that algorithm to map it on a GPU taking into account the different properties of the GPU.

As described in section 4.1.3.1 we determined two implementations of the algorithm on a GPU: the original and a pre-calculation algorithm. In this section we discuss the differences and how they are implemented on the GPU.

5.1.1 Original

The original algorithm is listed in 5.1

Listing 5.1: Time domain algorithm

```
1  Confidential
```

As we look at the memory requirements, this algorithm only requires the input and output matrices. This gives a total of $N_el \times N_el \times numberofsamples \times 4 + X_dim \times Z_dim \times 4 = 128 \times 128 \times 1600 \times 4 + 636 \times 221 \times 4 = 105.4MB$. Since each GPU has 896 MB this is not a problem. All necessary functions are also available on the GPU, so this algorithm is ready for implementation.

Programming for the GPU means that you have to write a kernel, a piece of code that is executed on the scalar processors, depending on the settings thousands of times, each thread with other parameters. For the time algorithm we have chosen that the kernel calculates one element of the result, meaning one pixel of the final image. This means that we have to produce $Z_dim \times X_dim = 636 \times 221 = 140556$ threads. Threads can be grouped in thread blocks, and thread blocks can be grouped in a grid. A block is executed on a single multiprocessor and a block can not have more than 512 threads. A multiprocessor can execute at most 8 blocks at a time, or a maximum of 1024 threads. To satisfy these constrains the output matrix of the algorithm is divided into blocks. Each element in that block(pixel) will be calculated by a thread. The pseudo code for the CPU will therefore be:

Listing 5.2: CPU code time domain

```
1  Copy sensor data to GPU
2
3  {
4    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
5    int x(X_DIM / threads.x);
```

```
6     if (X_DIM % threads.x>0) x++;
7     int y(Z_DIM / threads.y);
8     if (Z_DIM % threads.y>0) y++;
9     dim3 grid(x,y);
10
11    IWEX<<< grid, threads >>>(d_result, d_data);
12  }
13
14  Copy result from GPU
```

For the complete source code see appendix C. This piece of code creates the block grid and thread blocks as described and executes the kernel IWEX with those parameters. Figure 5.1 illustrates the division of the output matrix. The red line divides the elements into blocks which equal to thread blocks. Each thread will calculate one pixel.

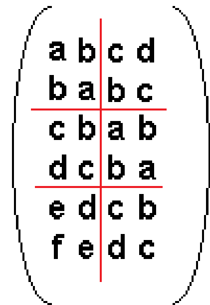


Figure 5.1: Output matrix divided into blocks (this case block size of 2)

The pseudo code for the kernel is listed in 5.3.

Listing 5.3: GPU code time domain

```
1  Confidential
```

5.1.2 Pre-calculation

The pre-calculation algorithm is listed in listing 5.4.

Listing 5.4: Time domain algorithm with look-up tables

```
1  Confidential
```

The first part of the algorithm, which does the pre-calculations, will be calculated on the CPU(Matlab). This process is not time critical and by doing so it does not have the precision limitations. The results of the pre-calculations will be copied on the GPU global memory. The memory requirements become $X_dim \times Z_dim \times N_el \times 4 + X_dim \times Z_dim \times N_el \times N_el \times 4 = 636 \times 221 \times 128 \times 4 + 636 \times 221 \times 128 \times 128 \times 4 = 9.3GB$. This can not be done on the current GPU's and therefore we have to modify the pre-calculations. As we look at the algorithm the t_i_n matrix needs the most memory. The

t_i_n matrix consists of a distance matrix for every loop iteration($N_el \times N_el$). The distance is from the transmitter element to a particular point in the metal and back to a receiving element. Figure 5.2 illustrates how the different matrices are constructed. In the figure are three matrices, all cases the transmitter is equal to the receiver. The difference between the matrices is which element is transmitting and receiving. The arrows represent the distance to a point. That distance is the value in the matrix at the location of that point. As you can see the distances in the three matrices are exactly the same for all points compared to each other. This means that if we store one of these matrices we can recreate all three. This property is not only for equal transmitters and receivers, but is also hold up for all transmitter receiver combinations. In listing 5.5 is the modified pre-calculations algorithm illustrated.

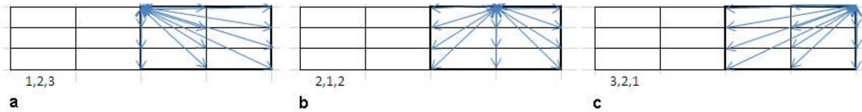


Figure 5.2: Vector distance of a)transmitter and receiver 1; b)transmitter and receiver 2; c)transmitter and receiver 3. With indexes to the original matrix

Listing 5.5: Adjusted pre-calculations

```
1  Confidential
```

This reduces the memory requirements to $2 \times N_el \times X_dim \times Z_dim \times 4 = 2 \times 128 \times 636 \times 221 \times 4 = 144MB$. It is advantageous for the GPU to put the t_i_n matrix twice in the memory as one big matrix. In figure 5.2 there are numbers at the bottom, these numbers represent the column of the first matrix. As you can see in the second matrix, the first column is the same as the first in the first matrix. The first matrix is the part stored in memory. If we just store the matrix(not twice) then first the second row is fetched from memory and thereafter the first row and then again the second row. Note that these fetches are all executed by different threads. To get the optimal performance of the memory, the fetches should be as streaming as possible. Meaning that fetching columns one, two and three in that order is quicker than two, one and two. That is why it is preferable to store each distance matrix also mirrored. The final memory requirement will then be $3 \times X_dim \times Z_dim \times N_el \times 4 = 215.9MB$.

The CPU code for the pre-calculation implementation is largely the same as the original version, the only difference is the number of parameters, because not only the sensor data and results are in global memory of the GPU but also the pre-calculation matrices. The kernel is listed in 5.6.

Listing 5.6: GPU code time domain with look-up tables

```
1  Confidential
```

As you can see in the pseudo code there are two index calculations. The first calculation is the index in the geometric spreading(GS) matrix. The matrices are stored by matlab in column major order. That would require an index calculation of `gs_index = bx*BLOCK_SIZE*Z_DIM+by*BLOCK_SIZE+ty+tx*Z_DIM+nti*Z_DIM*X_DIM;`.

The bx variable skips BLOCK_SIZE columns the by variable skips BLOCK_SIZE rows, the tx variable skips tx columns, the ty variable skips ty rows and the last variable nti skips a complete matrix of Z_dim times X_dim. The GPU executes from a block all the tx threads at the same time. The hardware tries to gather all memory fetches from those threads and coalesces it into as few as possible memory fetches. This means that when all threads are fetching memory locations next to each other the hardware will execute just one memory fetch (depending on bit width, etc). If we translate this property to the index calculation we want the form SOMETHING * ty + tx because threads are executed in the x dimension first. So the tx variable should just be added. In the previous formula the tx is multiplied by Z_DIM. This is necessary because of the column major order. If we switch to row major order we get a index calculation formula of `gs_index = bx*BLOCK_SIZE+by*X_DIM*BLOCK_SIZE+ty*X_DIM+tx+nti*Z_DIM*X_DIM;` which satisfies the previous property. The second index calculation is done for the tin matrix. The tin matrix is also stored in row major order. First of all remember that we used a property of the distances to decrease the memory. The drawback for that decision is a more difficult index calculation. First of all we have to calculate which pre-calculated matrix we need. This is done by $x = |nti - nri|$. The index is than calculated by:

```
if(nri<nti) {
   tin_index = X_DIM*Z_DIM*x+(X_DIM-nri*INVERS_XI_RES)+tx+
               ty*X_DIM+bx*BLOCK_SIZE+by*X_DIM*BLOCK_SIZE;
} else {
   tin_index = X_DIM*Z_DIM*x+(X_DIM-nti*INVERS_XI_RES)+tx+
               ty*X_DIM+bx*BLOCK_SIZE+by*X_DIM*BLOCK_SIZE;
}
```

The if structure is needed because we need the minimum of nri and nti. An if structure will decrease the performance dramatically if all threads in the same warp don't take the same path (divergent). If the threads do not take the same path, the hardware will first execute the threads of the first path leaving the others idle and when those finish with the divergent path the second set of threads will execute, leaving the first idle(sequential executing). INVERS_XI_RES is equal to $\frac{1}{xi_res}$ which equals the number of pixels between to adjacent piëzo elements.

5.2 Implementation frequency domain

Also for the frequency domain there are two major implementations possible: the original and an algorithm that pre-calculates the matrices.

If you look at the algorithm you can see that each iteration of the outer loop results in one line of results(pixels) in the z-dimension. The inner loop consists of two matrix calculations. Those matrix calculations require the most calculations. Therefore we will implement the algorithm such that the most processing power is available in the inner loop. This is done by creating a block of threads that is responsible for calculation one part of a row. This means that for every pixel of the result we create block size threads. So if we have a result matrix of 640×64 with a block size of 16, we must create $\frac{640}{16} \times 64 = 2560$ blocks each block containing $blocksize^2 = 256$ threads. By doing this we have for each matrix calculations a block of threads available to calculate the result. The CPU code looks like the following listing 5.7.

Listing 5.7: CPU code frequency domain

```
1  copy data to GPU
2
3  dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
4
5  int x(X_DIM / threads.x);
6  if (X_DIM \% threads.x>0) x++;
7  int y(Z_DIM);
8  dim3 grid(x,y);
9
10 IWEX<<< grid, threads >>>(data, result);
11
12 get result from GPU
```

To calculate the matrix multiplication we divide the calculation in block size blocks and do the matrix multiplication on those sub-matrices. In this way each thread in a block can calculate one element of the multiplication. The different results of the sub-matrices have to be added to each other to get the final result(see figure 5.3). In the figure the first iteration loads and multiplies the red sub-matrices and the second iteration loads and calculates the blue sub-matrices. Adding the sub-result of the iterations will give the final result of the sub-matrix.

The advantage of using sub-matrices is that we can use the shared memory of the GPU to calculate these matrices. The shared memory of a GPU is shared among all threads on a single multiprocessor and has a size of 16kB. Fetching from shared memory is, if well organized, as fast as a register access, therefore it is advantageous if you handle the data more than once to use the shared memory.

5.2.1 Original

Mapping the original algorithm on the GPU means that only the data and the result are transported to the GPU and the GPU itself calculates the matrices. The GPU code looks like the following:

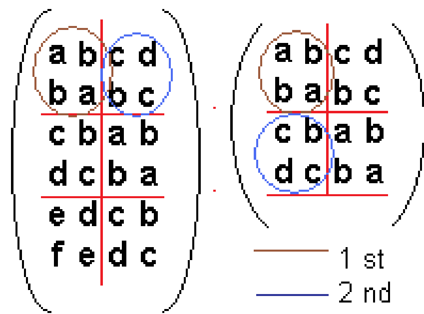


Figure 5.3: Sub matrices multiplication with the first and second iteration (r_itt)

Listing 5.8: GPU kernel for the frequency domain

```
1  Confidential
```

The code above is a CUDA code with pseudo code. There are some terms that are not handled yet. __global__ means that that function is an entry point for the CPU. __shared__ means that the declared variable resides in the shared memory of the multi processor.__syncthreads() is a function that synchronizes the threads in one thread block. The code first iterates over all frequencies, then over all columns of the matrix and then over all rows of the data. Each thread loads one matrix element(P_temp) of the global memory and stores that element in the shared memory. Each element consists of a real and an imaginary part. The thread then also calculates one element of the matrix and stores it also in the shared memory. If this is done all threads should be finished before continuing, this is done by the function __syncthreads(). This function blocks the thread from execution until all threads in that block have executed that function. If the threads are released each thread calculates one element using a complex matrix multiplication. After the matrix calculation we have to synchronize again before loading new data in the shared memory. If the row loop ends we have the result of the first matrix multiplication of sub-matrices.
The second matrix multiplication is not really a matrix multiplication. The algorithm specifies that we only need the diagonals of the result. That is why we only need to calculate the diagonals. From that diagonals we also only need the real values. For that reason each thread only has to calculate one war element, that element is not needed by another thread so does not have to be shared. And for that element we only have to calculate the real value. This is done in the next two lines in the kernel. First each thread calculates a war element and multiplies it with the results of the first matrix multiplication and stores that result in the shared memory. The elements have to be added to get the result of the second multiplication and finally that result is stored in global memory.
The sensor data is stored in global memory, row major order. But the original algorithm takes a subset of the data, namely the data from one frequency. Therefore if we want to create a fetch construction of SOMETHING*ty+tx we need to rearrange the data not only from column to row major order but also shuffle one row. If you look at the

algorithm, the first three elements in the x-dimension of P_data is equal to the first, 33, and 65 elements of the sensor data in the x-dimension. The next listing shows a function that copies the data from column major order to the wanted order.

Listing 5.9: CPU code for rearranging the sample data

```
1  unsigned int i(0);
2  for(unsigned int z(0);z<N_FREQUENCIES;z++) {
3    for(unsigned int y(0);y<N_EL;y++) {
4      for(unsigned int x(0);x<N_EL;x++) {
5        int j = x*N_EL*N_FREQUENCIES+y*N_FREQUENCIES+z;
6        d[i] = p[j];
7        i++;
8      }
9    }
10 }
```

This results in an index calculation formula of `f*N_EL*N_EL+r_itt*N_EL*BLOCK_SIZE+c_itt*BLOCK_SIZE+ty*N_EL+tx;` The calculation of the war elements is also extensive. The listing below shows the calculations.

Listing 5.10: GPU kernel code to calculate a W_A_r element

```
1  Confidential
```

These calculations are done in every thread in the most inner loop before the matrix calculations. There they have a major influence on the performance.

The last part of the kernel is addition of the partial result obtained by the last matrix multiplication. The addition is done with a algorithm of logarithmic complexity and is listed below.

Listing 5.11: GPU kernel code for adding the partial results

```
1  unsigned int s=BLOCK_SIZE/2;
2  for (unsigned int i(0); i<LOG_BLOCK_SIZE; i++) {
3    if (ty < s) {
4      data_r(tx,ty) += data_r(tx,ty + s);
5    }
6    s>>=1;
7    __syncthreads();
8  }
```

As stated before an if statement does not give the best performance when the paths are different within the threads in a warp. This if statement always gives that problem, meaning that this loop will always run in a sequential manner. The calculations are done in shared memory, the shared memory has 16 banks. To fully optimize the bandwidth to the shared memory, all the threads in a half warp should address another bank. If this does not happen a bank conflict will occur and the fetches are handled sequentially. This is the reason why a thread does not fetch ty and ty+1 but ty and ty+BLOCK_SIZE/2 in the first iteration. This will result in a first fetch of all sequential number 1 to 16, and

the second fetch from 17 to 32 if BLOCK_SIZE is 32 in the first iteration. Otherwise the fetches of the odd numbers are fetched the first iteration and gives a bank conflict at every bank for threads ty and ty+8. For the final result we have defined BLOCK_SIZE as 16. Therefore we have to add 16 partial results in one dimension. If we use the algorithm above, only eight threads are really executing the add instruction. Remember that the warp size is 32, meaning that all threads in that warp will always be synchronized by architecture. Therefore we can simplify the above add algorithm to the next algorithm.

Listing 5.12: GPU kernel code that is more optimal for adding the partial results

```
1 data_r(ty,tx) += data_r(ty,tx + 8);
2 data_r(ty,tx) += data_r(ty,tx + 4);
3 data_r(ty,tx) += data_r(ty,tx + 2);
4 data_r(ty,tx) += data_r(ty,tx + 1);
```

5.2.2 Pre-calculation

The frequency domain algorithm can also be implemented such that the W_A_r matrices are pre-calculated, so that the critical path of the algorithm has as little calculation as possible. The pre-calculation algorithm is listed in listing 5.13. The pre-calculation calculates the W_A_r matrices for every possible iteration and stores them in memory. The size of the memory therefore should be $Z_dim \times F_dim \times X_dim \times N_el \times 2 \times sizeofdatatype = 221 \times 150 \times 631 \times 128 \times 2 \times 4 = 21.4GB$.

Listing 5.13: Frequency domain algorithm pre-calculations

```
1 Confidential
```

The amount of memory is not available on the GPU. The W_A_r matrices have a toeplitz structure [15], meaning that a matrix can be reconstructed only from it's first column, therefore it is not necessary to store the complete matrix in every iteration, but just the first column. The pre-calculations with the use of toeplitz structure is the same but only line 11 changes to `W_A_r_n(:,counter)=W_A_r(:,1);`, and of course the memory allocation. The required memory will now be $Z_dim \times F_dim \times X_dim \times 2 \times sizedatatype = 221 \times 150 \times 631 \times 2 \times 4 = 167MB$

The GPU code is very similar to the original algorithm, the only difference is that instead of calculating the war elements, it will be loaded from memory. The index calculation is done using the code.

```
int war_index = INVERS_XI_RES*ty-(tx+bx*BLOCK_SIZE)+
                r_itt*BLOCK_SIZE*INVERS_XI_RES;
war_index = fabsf(war_index);
war_index+=f*X_DIM+by*N_FREQUENCIES*X_DIM;
```

The first two lines calculates the index of the correct matrix element with the use of the toeplitz structure, the last line is needed to select the right column of stored data. Note that the pre-calculation matrix is stored in column major order in the GPU memory.

5.3 Optimizations

Remember that there are two caches on each multiprocessor, a constant cache and a texture cache. The constant memory is not used explicitly in the code because it is not large enough, but is used by the compiler to store constants which are declared with defines in C. The texture cache can be used if variables are declared as a texture. Or better said, a piece of global memory can be binded as texture memory. The texture cache is optimized for 2D matrices. We use this matrix for the data as well as for the W_a_r pre-calculated matrices. Both matrices have the property of temporal and spatial locality. The data is used by multiple blocks running on the same multi-processor. The pre-calculated values have their locality by a single thread.
Another optimization is in the shared memory. As it can be seen in the implementation, there are four variables declared in the shared memory, the data and the war sub-matrices. But the real and imaginary parts are separated. This is because if we declared a type so that the real and imaginary parts are stored after each other in the memory, it will create bank conflicts in the shared memory.tx=1 and tx=9 would both request data from bank 1, the same is true for 2 and 10 etc.

6 Results

In this chapter we discuss the results and compare those to the expectations. For the benchmarks we have set up a machine with a core i7 975 3.3GHz CPU on a Asus P6T Deluxe V2 motherboard. The machine has two GTX295 GPU cards each consisting of two GPUs. The motherboard has two 16 lanes PCI Express 2.0 slots and supports up the 36 lanes. All the benchmarks have the parameters as described in table 6.1 if not specified otherwise.

Parameter	Value
block_size	8
1/zi_res	5
1/xi_res	5
z_dim	64
xi_roi_start	1
xi_roi_end	32
ROI	32
N_SAMPLES	1600
F_dim	150
N_el	128
Number of GPUs	4

Table 6.1: The values used for the benchmarks if not specified otherwise

6.1 Time domain

For the time domain we have discussed two implementations. A pre-calculated version and the original version, where all calculations are done on the GPU.

6.1.1 Original algorithm

Figure 6.1 illustrates the time needed for one execution of the original time domain algorithm implemented on a GPU, depending on the number of elements.
The time is compared to the number of FLOPs that each kernel has to calculate. As discussed earlier the number of FLOPS is of the order $O(N_el^2)$. As you can see the execution time looks very similar. This can be deceiving because the number of fetches of each kernel to the global memory is also in the order of $O(N_el^2)$. But because of the arithmetic intensity of the t_i_n element the bottleneck will be the calculation instead of memory. The red line illustrates the time needed for transporting the sample data to and the result from the GPU. The size of the sample data is dependent of the number of

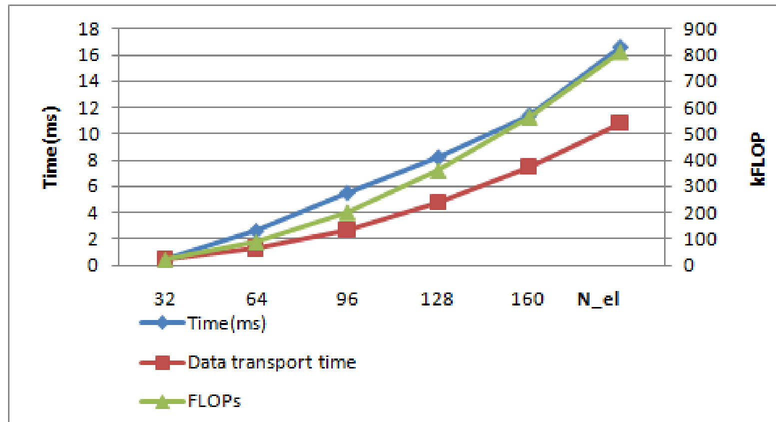


Figure 6.1: Time and FLOPs versus N_el of the original time domain

elements and is of the order $O(N_el^2)$. The time it takes to copy the data to and from the GPU is added to the execution time in all the next benchmarks.

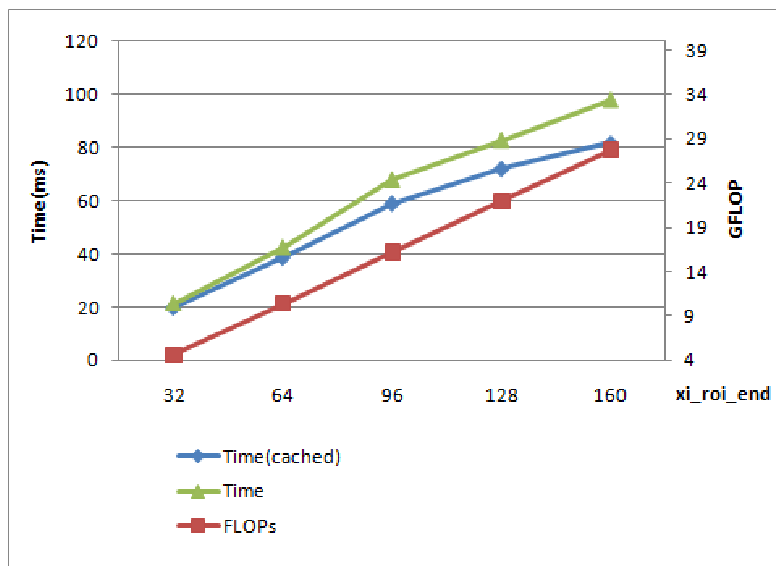


Figure 6.2: Time and FLOPs versus ROI of the original time domain

In figure 6.2 the execution time is displayed depending on the Region Of Interest. If the ROI doubles the only thing that will change is the number of thread blocks executed on the GPU. Since the ROI only changes the x-dimension, the number of blocks is linear with the ROI, and therefore the execution time is also linear depending on the ROI and also the number of FLOPs needed for one image(red line). In the figure there are three lines. The first is the original algorithm loading its data from global memory. The second line loads its data using texture fetching. The use of texture is preferred because each multi-processor in the GPU has a texture cache. The reason why the cache is more optimal is that different blocks within the same processor need the same subset of data.

The third line illustrates the needed FLOPs.

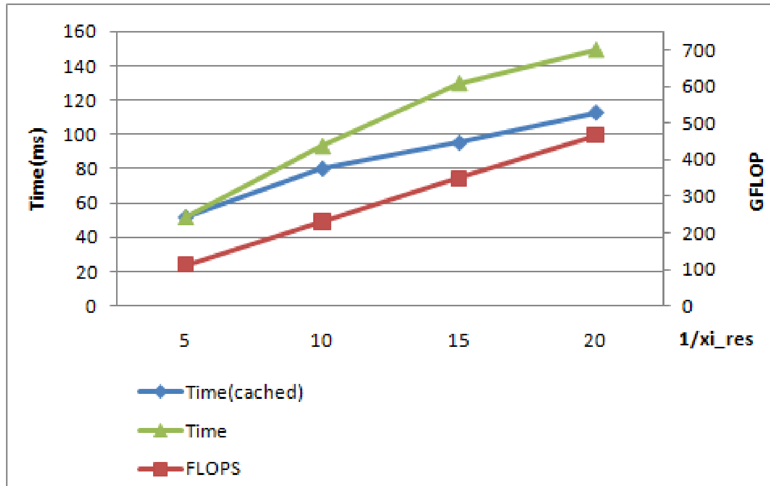


Figure 6.3: Original time domain algorithm versus resolution

Figure 6.3 shows the execution time versus the resolution in the x-dimension. Again the figure shows the difference between the cached and the non cached version and again the cache has the preference, because if the resolution is increased different blocks are more likely to use the same data. The increase in resolution in one direction is linear to the number of calculations and also in the number of blocks executed on the device. The variation in resolution in the z-dimension respond exactly the same. Meaning that if both resolutions are changed at once, the calculations and execution time will increase in a quadratic manner.

6.1.2 Pre-calculation

The second algorithm we tested is an algorithm that pre-calculates the t_i_n matrix. Therefore each kernel loads one t_i_n element and one data element for each iteration of the transmitter and receiver combination. Because the number of FLOPs and the number of memory fetches is of the same order, the response of this algorithm is the same as the original with respect to the number of elements, region of interests and resolution. But this algorithm is memory bounded, meaning that a kernel spends most of its time waiting on data from global memory. In figure 6.4 the comparison between execution times is illustrated. The original algorithm is the quickest and increases slower than the pre-calculated algorithm.

6.2 Frequency domain

Also for the frequency domain we have implemented two versions. The first version is most equal to the original algorithm and the second version is an implementation that uses pre-calculated matrices that are stored in global memory. Figure 6.5 illustrates the difference in execution time between the two implementations. As you can see this

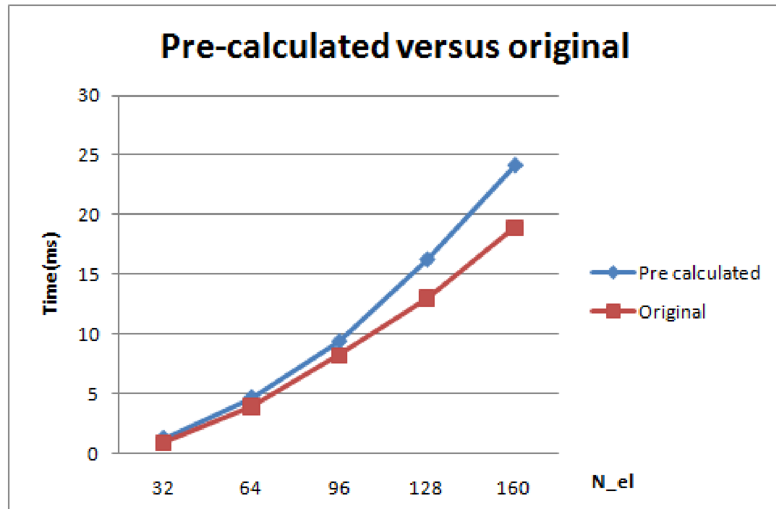


Figure 6.4: Original time domain algorithm versus pre-calculated

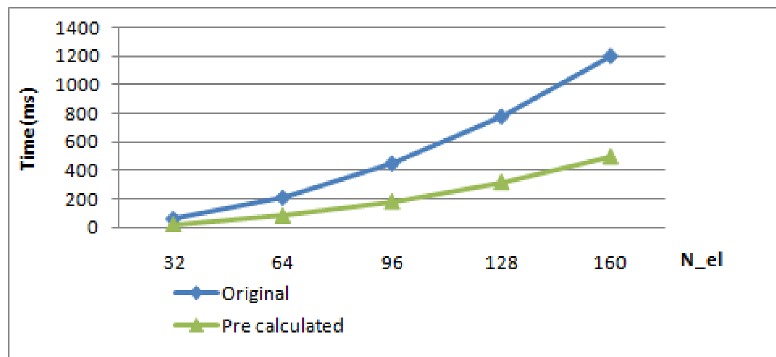


Figure 6.5: Original frequency domain algorithm versus pre-calculated

time the original implementation is the slowest. If we look at the difference between the pre-calculated implementation and the original implementation, we try to calculate one war element instead of loading it from memory as is done in the pre-calculation case. But the problem with the original algorithm is that it uses more registers than the pre-calculated implementation. Since the number of registers is bounded, the compiler stores the data in the local memory of a kernel. But the local memory does not reside in on-chip memory but in global memory. The difference in local memory is 120 bytes which is 30 registers. So instead of pre-calculating and loading 8 bytes of data, we now have to load all the local memory which is 120 bytes at each kernel each iteration(Z_dim and F_dim).
The original implementation and the pre-calculated implementation respond equally to deviation in N_el, ROI and resolution, therefore we are only discussing the pre-calculated version. In figure 6.6 the execution time is illustrated as a function of the number of elements compared to the number of FLOPs needed for one execution. The number of FLOPs is of the order $O(N_el^2)$ and as we can see the execution time is similar.

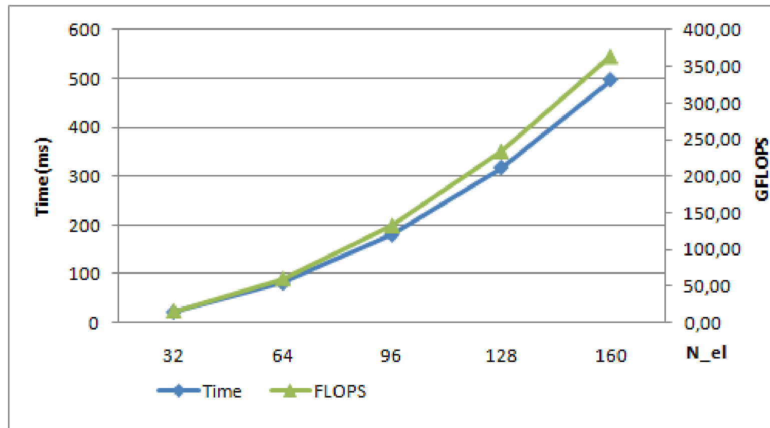


Figure 6.6: Execution time versus number of elements

The implementation is also bounded by the required calculation instead of memory bandwidth. There are enough threads available to fill up the time needed for global memory loading.

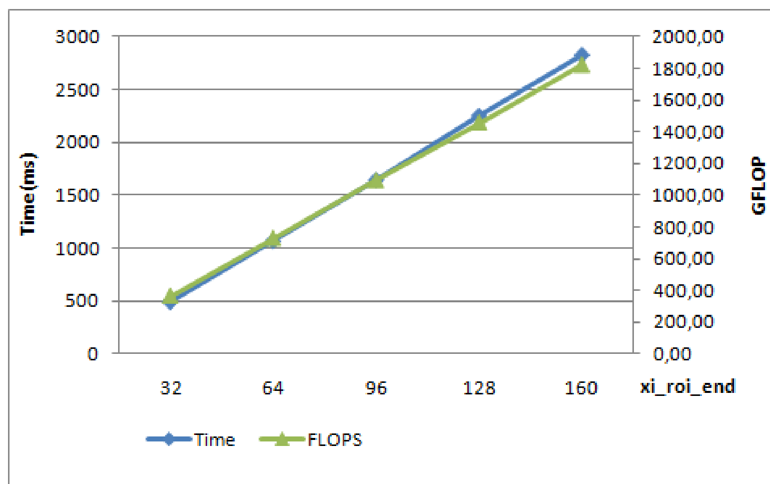


Figure 6.7: Execution time versus the region of interest

The region of interest determines the width of the x-dimension. The number of thread blocks are linear with the width of that dimension, meaning that the execution time will also be linear as is illustrated in figure 6.7. The resolution parameter xi_res also determines the width of the x-dimension and therefore has the same influence on the execution time which is again linear as can be seen in figure 6.8. .

All results presented so far are based on execution on four GPU's. The execution times are based on the GPU which took the longest to finish the algorithm. The GPU which took the longest is always the same in this case and that is the GPU that has a monitor connected to it. A GPU used to serve the primary monitor can be used to run CUDA programs, but the GPU will switch from CUDA mode to graphics mode used by

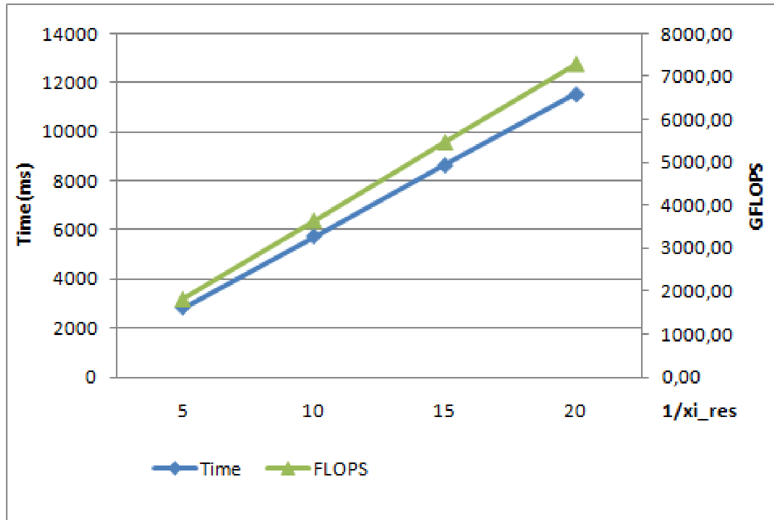


Figure 6.8: Execution time versus the resolution

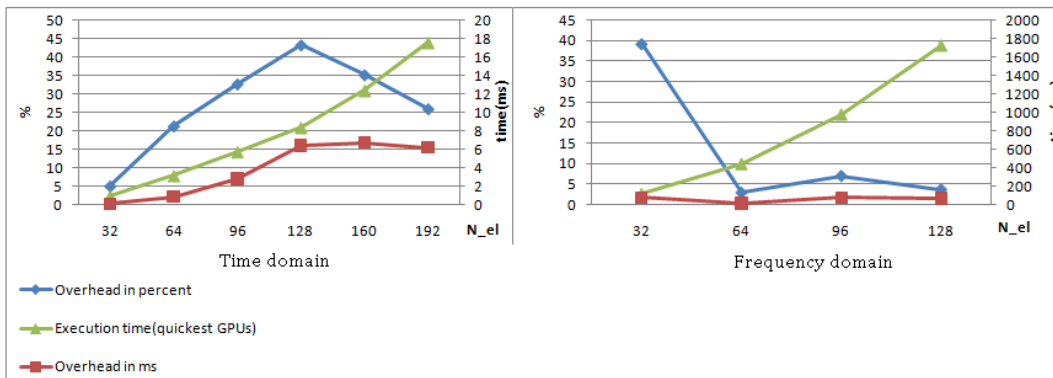


Figure 6.9: Monitor overhead for the time and frequency domain

Windows, therefore the execution time is higher than you should expect for a single GPU. Figure 6.9 illustrated the overhead in percent and milliseconds. As can be seen that the monitor overhead has a large influence on the execution time in the time domain. But if we look at the frequency domain at the higher regions the overhead stabilizes around 5 %. The overhead is determined by the execution of the algoritm using four GPUs with equal load. The differences in execution times is counted as overhead.

In figure 6.10 the difference in execution time is illustrated and also the speedup if you scale from one GPU to four GPUs. As you can see the speedup reaches four which should be the optimal. The small deviation from four is due to the data transport to and from the GPU. Note that every GPU calculates a sub result. All the results obtained from the different GPU's should be added with each other, this is done on the CPU. This time is taken into account in the above figures. But it illustrates that the CPU is also necessary for the final result if executed over multiple GPU's.

If we look at the number of operation per second(FLOPS) for a single GPU, based on

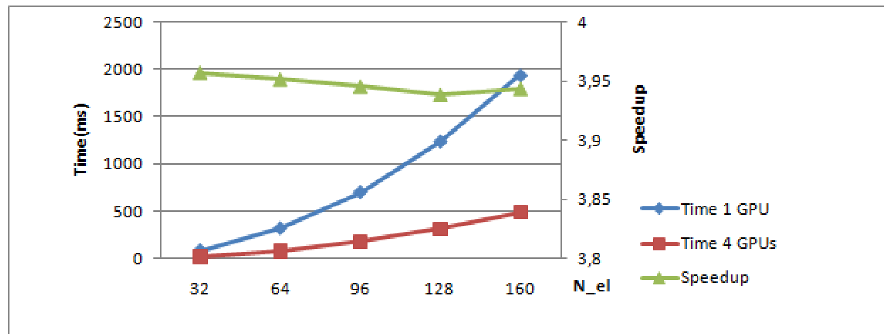


Figure 6.10: Execution time 1 GPU versus 4 GPU's

the FLOPS needed for one iteration, a GPU calculates 184 GFLOPS. Since each GPU can have a theoretical FLOP count of 933 GFLOPS the results do not seem optimal. Note that we predicted a maximum of 379 GFLOPS for each device. If we execute the implementation with the parameters expressed in table 3.1 on a single GPU, what we used for the exploration part in this thesis, the execution takes 15 seconds. This is close to what we expected but due to optimalizations of the algorithm this eventually is two times slower than expected. The reason why the performance is lower is that the overhead needed for the calculation of the indexes is higher than expected. Nevertheless the speedup compared to the CPU is greater than expected.

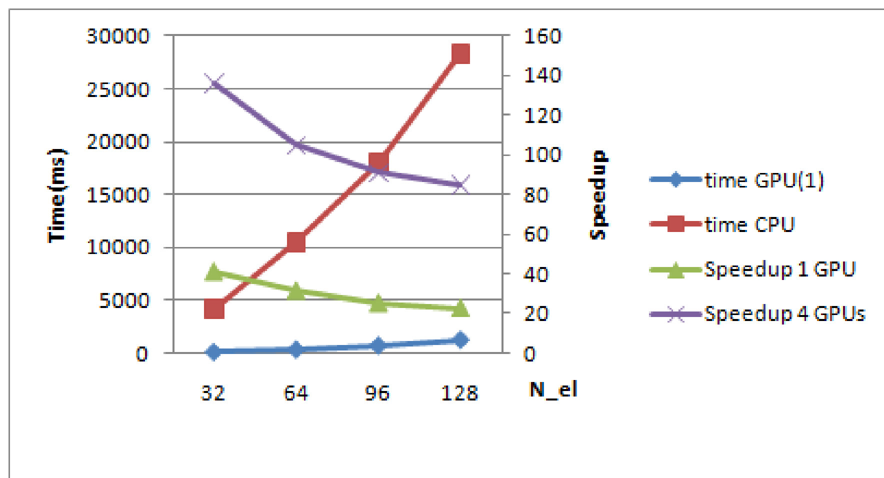


Figure 6.11: CPU versus GPU

Figure 6.11 illustrates the speedup of the GPU compared to the CPU depending on the number of elements. As you can see the speedup with one GPU is stabilizing around 20 and with 4 GPUs around 80. The speedup is more as we decrease the number of elements, this effect is caused by the CPU which is less optimal when using smaller matrices. Note that by all of these results the time of the GPU consists of the kernel time, the transportation time and the time the CPU needs for arranging the data.

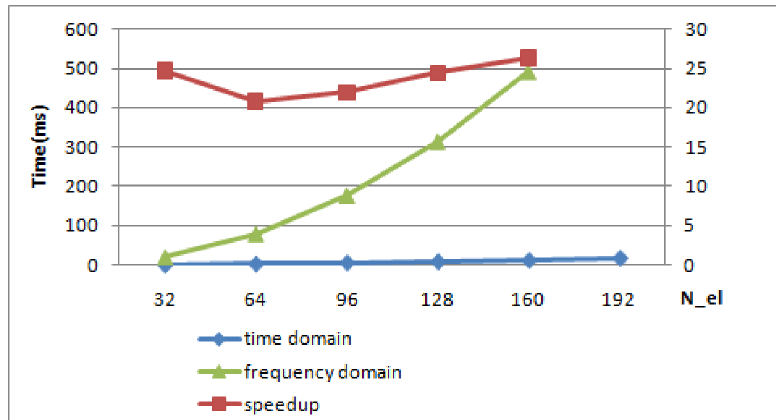


Figure 6.12: Time domain versus frequency domain

Figure 6.12 shows the difference in execution time between the frequency domain algorithm and the time domain algorithm. The figure illustrates that the time domain is always quicker if we look at the execution time of the GPU(including transporting time).

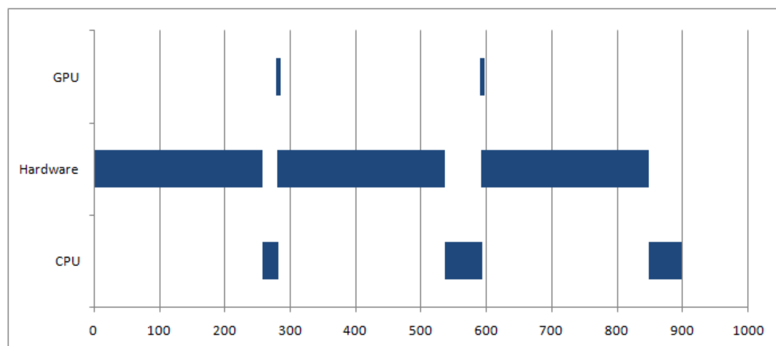


Figure 6.13: System performance using time domain with 64 piëzo elements

The previous benchmarks are based on input and output files. But the algorithm can be used with acquisition hardware. To be able to use the hardware Matlab functions for the GPU are implemented. With this GPU-Matlab functions in combination with existing function to control the acquisition hardware a test system is made. Figure 6.13 illustrates the sequence and the different execution times. First the acquisition hardware is triggered to get the sample data. The hardware send it to the CPU and it becomes available in Matlab. After that the samples are copied to the GPU and the GPU is triggered to execute the algorithm. The trigger is non-blocking and for that reason the hardware can do another acquisition while the GPU is busy. After the GPU and acquisition hardware are both done, the result from the GPU can be copied back to the CPU and Matlab, and Matlab creates a picture from that result. If that is done the new samples are copied to the GPU and again the GPU and hardware can be triggered for new data. To create figure 6.13 we used 64 piëzo elements in combination with the time

domain (xi_roi_end is 64 and 1/xi_res is 5). As can be seen in the figure the most time is lost in the acquisition hardware, in this case the GPU is idle most of the time. So if we want more images per second we have to optimize the acquisition hardware. In this case the sequence frequency (time between two firings) is currently 250 Hz and slows the whole system down.

7 Conclusions and future work

7.1 Conclusions

As discussed in the platform exploration chapter the FPGA in combination with fix-point calculation will calculate the IWEX algorithm in the least amount of time compared to the other three discussed platforms. If floating point operations are needed a FPGA has less performance. In that case a GPU is expected to be the fastest.
For this thesis we have implemented the IWEX algorithm on the GPU platform, resulting in a speedup of over 20 times compared to a quad core CPU. Regardless of the speedup the theoretical(usefull for the algorithm) FLOP count is 184 GFLOPS of the maximum of 933 GFLOPS for one GPU.
Not all algorithms are suitable to map on a GPU. A GPU needs a massive amount of independent threads so that each scalar processor can do other useful work while the threads are waiting on the data. Also the number of calculations per fetched byte from global memory should be large.
The relatively slow bandwidth of the PCI-Express 2.0 16x is in this case not a problem because the algorithm takes much more time than the transport of the data. Besides there are some options to overlap the transport with the calculations.
If we compare the execution time between the time domain and frequency domain, the time domain will always be faster then the frequency domain even though the data transport takes longer.
CUDA and the programming model are easy to understand for a C programmer, but it is more difficult to get an optimal mapping for the GPU. This already arises the question whether there is an optimal solution. Before it is implemented it is hard to estimate the performance of the mapping. Register usage is difficult to predict but very important to utilize the GPU.
If we look at the complete system then currently the acquisition hardware has become the bottleneck, therefore if more speedup is required the acquisition hardware should be looked at. The problem lies in the sequence frequency (time between two firings) which is currently 250Hz. Changing this frequency is not part of my research and can be dependent of the material properties.

7.2 Remarks

Nvidia's CUDA is free to use, and therefore anybody with an CUDA capable GPU can create programs without any initial costs. Nvidia is optimizing the compiler and it's GPUs for general purpose usage. At this moment debugging is not supported, there is an emulation mode but that compiles the program for CPU and does not emulate the GPU device. Also emulating one execution takes much more time than the execution

on the GPU. Also the emulator runs one block at a time, so races conditions between blocks are not noticed. Nvidia supplies a program called visual profiler which can count different events on the GPU in runtime. But that program is still in development and is usable, but not optimal and also not all devices support all events. But it is a start to get an insight of how your program is performing on the GPU.
Another disadvantage of the GPU system is that the architecture of the GPU is not documented. The programming manual describes different considerations. But there is no detail design of all the elements. This makes optimization difficult together with the different parameters that are all linked to each other. Therefore you should run your application every time you think it is optimized.

7.3 Future work

To further optimize the IWEX application it is important to first know what the range of the parameters is. When the parameters are known the time needed for calculation of the algorithm can be estimated. The next optimization is that every step in the process should be overlapped(pipelining). So the acquisition should be acquiring data for the next iteration when the GPU is busy calculating the current one. The same equals for the calculation of the FFT (if needed) for the transportation to and from the GPU, and for the addition of the partial results from the GPU. Currently matlab is creating a picture from the result of the algorithm. It is possible to use the GPU to cooperate with OpenGL and use OpenGL to create and render the image on the monitor.
If we look at future specifications of CUDA I would like to be able to specify which multi-processors to use for a specific kernel, as well as to run multiple kernels on the different multi-processors. In that way we can assign multiple tasks on a GPU and it has also advantages for the caches in the multi-processors. Therefore we also need the ability to synchronize between the kernels (or blocks) on different multi-processors. Note that the ability to cooperate with OpenGL is not the same because graphic mode and CUDA mode are not running at the same time, but are time sliced controlled by the device driver.
It is possible to extend the IWEX algorithm to calculate 3D pictures[15]. To create these pictures more data is necessary as well as more computations. I think it is possible to produce 3D pictures with the GPU of course depending on the maximum time allowed to produce one picture and the number of GPU's in the system. Because the delay of the PCI Express bus is negligible it is possible to put more GPUs in one system with the drawback that each GPU will only have eight PCI lanes available for communication.

Bibliography

[1] Altera, *Designing and using fpgas for double-precision floating-point math*, http://www.altera.com/literature.

[2] ———, *Virtex-5 fpga user guide*, http://www.altera.com/literature, 2009.

[3] Dave Altiva, *Intel p35 bearlake motherboard*, http://hothardware.com/.

[4] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ort, and Gregorio Quintana-Ort, *Exploiting the capabilities of modern gpus for dense matrix computations*, 2008.

[5] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata, *Cell broadband engine architecture and its first implementation*, http://www.ibm.com/developerworks/power/library/pa-cellperf/.

[6] Chess, *Iwex user manual*, 2008, internal document, Chess, Haarlem, The Netherlands.

[7] Clearspeed, *Clearspeed*, http://www.clearspeed.com/.

[8] Sony Corporation, *The basics of cell computing technology*, 2008.

[9] Genna Cummins, Dr. Rob Adams, and Theodore Newell, *Scientific computation through a gpu*, 2008.

[10] Rick C. Hodgin, *High-end coprocessors: An overview*, http://www.tgdaily.com/.

[11] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek, and Jack Dongarra, *The playstation 3 for high performance scientific computing*, 2007.

[12] Wen mei Hwu, *Programming massively parallel processors*, http://courses.ece.uiuc.edu/ece498/al/Syllabus.html.

[13] NVIDIA, *Accelerating matlab with cuda using mex files*, 2007.

[14] ———, *Programming guide 2.0*, 2008.

[15] Niels Pörtzgen, *Imaging of defects in girth welds using inverse wave field extrapolation of ultrasonic data*, Ph.D. thesis, TU Delft, 2007.

[16] Rys, *Nvidia g80: Architecture and gpu analysis*, "http://www.beyond3d.com/".

[17] Larry Seiler, Doug Carmean, and Eric Sprangle, *Larrabee: A many-core x86 architecture for visual computing*, 2009.

[18] Tilera, *Tilepro64 overview*, http://www.tilera.com.

[19] Unknown, *bandwidth of pci express to and from the gpu*, 2008.

[20] Roger van Schie, *Master thesis assignment*, 2008, internal document, Chess, Haarlem, The Netherlands.

[21] Richard Walsh, Steve Conway, Earl C. Joseph, and Jie Wu, *Powerxcell*, 2008.

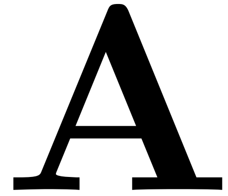
[22] Xilinx, *Floating-point operator v4.0*, `http://www.altera.com/literature`, 2008.

[23] Claudiu Zissulescu, *Report 1: Iwex algorithm: floating point to fix point conversion*, 2008, internal document, Chess, Haarlem, The Netherlands.

[24] ______, *Report 2: Iwex algorithm: a data-flow analysis approach*, 2008, internal document, Chess, Haarlem, The Netherlands.

[25] ______, *Report 3: Iwex algorithm: Architecture issues and solutions*, 2008, internal document, Chess, Haarlem, The Netherlands.

Time domain algorithm



Confidential

B Complex numbers

Throughout this document we have used complex numbers and assumed that it takes 6 FLOPs to multiply two complex numbers. Well in fact this can depend on the technology. If we take a normal complex multiplication: $(a + bi)(c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i$. Then we need 4 multiplications and 2 additions which will come down to 6. But the number of multiplications can be reduced to three in different ways which are all alike: $(a + bi)(c + di) = (ac - bd) + ((a + b)(c + d) - ac - bd)i$ $(a + bi)(c + di) = ((a - b)(c + d) - ad + bc) - ad + bc)i$

But this has the disadvantage of more additions(total of 5).

Time domain implementation code

C

C.1 CPU code

```
Confidential
```

Frequency domain algorithm D

Confidential

E Frequency domain algorithm code

E.1 CPU code

Confidential

F MATLAB commands introduction

Matlab makes no distinction between a single value or matrices. A single value is a matrix of dimensions 1 by 1.

C=A.*B	$c_{ij} = a_{ij} b_{ij}$
C=A*B	$c_{ij} = a_{i1} b_{1j} + a_{i2} b_{2j} + ... + a_{in} b_{nj}$
C=A.^x	$c_{ij} = (a_{ij})^x$
C=A./B	$c_{ij} = \frac{a_{ij}}{b_{ij}}$
C=s:e	$v_{1j} = s + (j-1)$ where j runs from 1 to $(e - s + 1)$
C=s:f:e	$v_{1j} = s + (j-1)f$ where j runs from 1 to $(\frac{e-s}{f} + 1)$
C=A(:,x)	C consists of all the rows of the columns indicated by x
C=A(x)	$C = a_x$
C=A(i,j)	$C = a_i j$
C=A(B)	$c_{ij} = A(b_{ij})$
C=A.'	$C = A^T$(not conjugate)
size(A,x)	returns the size of the x dimension of A
round(x)	returns x rounded to the nearest integer.

G Prototype

A prototype was already developed before it became a thesis project. It is useful to look at the prototype to see what the IWEX algorithm eventually is meant to do, and the practical aspect of it. But of course also to see how the system performs with that specifications.

This chapter will not discuss the irrelevant details. Please read the architecture documents(should here be a ref?).

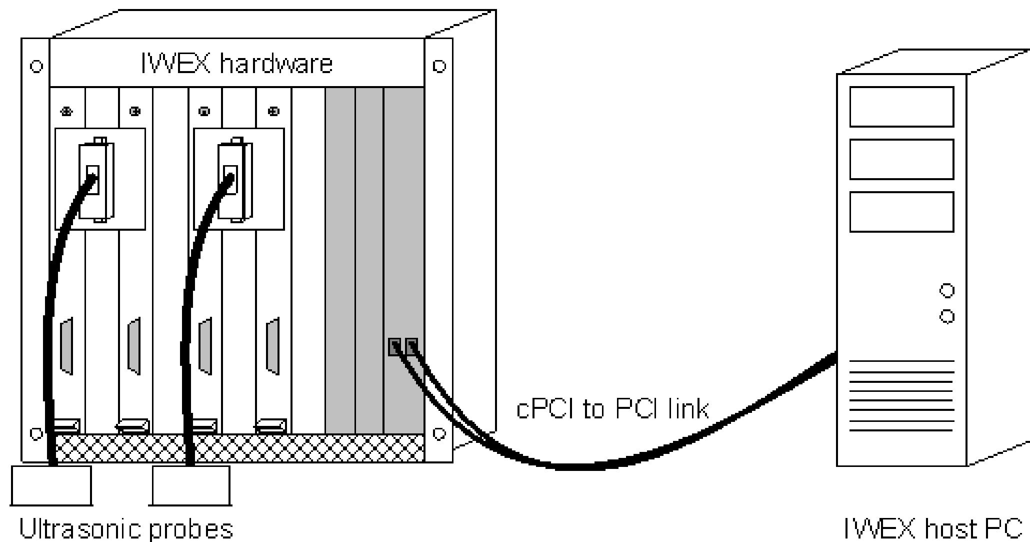


Figure G.1: IWEX system overview

An overview of the system is illustrated in figure G.1. The system consists of two ultrasonic probes of 64 elements each. Each 32 elements are connected to a IWEX hardware board which contains hardware to trigger the elements but also filter the result. For each channel 1600 samples are taking and led to a FPGA which will calculate the fast Fourier transformation for each channel. We now have got, for each channel, 1600 frequencies , we only need 150 and those are send to the host PC. The host PC will receive that and calculate the image.

In more detail: Triggering or firing the elements is out of scope for this document. But after the triggering comes the receiving and that is done at 12bit. Every element is triggered one after the other and all elements receive 1600 samples after every triggering. This means that if we take 128 elements we receive for one frame $128 \times 128 \times 1600$ samples of 12 bit which is 26214400 samples and 314572800 bits or 37.5 MB. The data receives the FPGA and the FPGA will do the FFT and truncate the precision to 8 bits. According

to [23] this will not lead to a unexceptable fault in the end image. The data stream from the FPGA is(again 128 elements) $128 \times 128 \times 150$ samples which is 2457600 complex samples each consisting of a 8 bit real and imaginary part. So the total size of one frame is reduced to 4.7 MB. This data is send to the host PC using a PCI link(DMA). A practical benchmark is done and the throughput was about 100Mbyte/s sustained data rate([6]). Once the data is on the host PC, which consists of a Intel Core2Quad Q9450 4x2,66 GHz 12M Box with 2 x 2 GB DDR2800, the frequency domain algorithm is used to calculate the final image.

The algorithm is only implemented for 64 elements using a region of interest(ROI) of 60 elements. As calculated in chapter 2 we need 102 GFLOPS for one image. The memory required by the implemented algorithm depends mainly on the two partial look-up tables and the input data matrix G.1 and is around 50 MB.

G.1 IWEX algorithm: Memory and computation requirements

The algorithm what is currently implemented on the quad core is the following(G.1):

Listing G.1: Frequency domain algorithm

```
1 Confidential
```

G.1.1 Computation requirements

For the computational requirements the algorithm can be reduced and altered to be ported to code:

```
1 Confidential
```

The order of this algorithm is $\mathcal{O}(n^5)$. To define the total number of operation we have to remember that the calculations are complex(except the lut). Each complex multiplication takes 4 multiplications and 2 additions and a complex addition takes two additions. We define $N_z = \frac{z_end - z_start}{z_res} + 1$ and $N_f = \frac{i_f_end - i_f_start}{step_f} + 1$ So the number of operations are:

$$\begin{aligned} N_{op} = \quad & N_z(N_f(2N_xN_el + 6N_xN_el \\ & +6N_xN_el + 8N_xN_elN_el + 4N_xN_el) \\ & +N_xN_x) \end{aligned} \tag{G.1}$$

So for example: $N_el = 64; X_end = 276; N_z = 74; N_f = 150$ then the number of operations is 104 GFLOPS per frame(5.3TFLOPS for values of 3.1). Note that this is the lower bound number. Instructions need to be added depending on the platform. If we take the quad core. The processor needs to load the data into registers before the multiply can be handled. This is a critical task for the programmer and compiler, to fully optimize the inner loop(or maybe even loop unrolling and other optimizations). Note that these supporting instructions depend on the platform and it is possible that these instructions can be done in other execution units.

Matrix	Dimension	Dimension in bytes	Size in Bytes N=64;ROI=60
W_A_r	N_x x N_el	N_x x N_el x 8 x 2	276 k
partial1_W_A_r	N_x x N_el x N_z	N_x x N_el x N_z x 8 x 2	20.0 M
partial2_W_A_r	N_x x N_el x N_z	N_x x N_el x N_z x 8 x 2	20.0 M
p_prop_p_p	N_f x N_x	N_f x N_x x 8 x 2	646 k
p_prop_p_p_real	N_f x N_x	N_f x N_x x 8	323 k
roi_p_p	N_z x N_x	N_z x N_x x 8	159 k
P_data	N_f x N_el x N_el	N_f x N_el x N_el x 8 x 2	9.4 M
W_A_r_partial2_next	N_x x N_el	N_x x N_el x 8 x 2	276 k
lut_sqrt	N_f	N_f x 8	2 k
total			49.5 M

Table G.1: Sizes of matrices dependencies

G.1.2 Memory requirements

G.1.2.1 Size

If we take the calculations regarding the memory requirements of report three[25] and if we take N_el=64; X_end=276; N_z = 74; N_f = 150 then the total memory needed is 2.1 GByte(ROI=20 and constant). If we extent this to 128 elements the required memory is 4.3 GByte which is rather large. But this calculation is based on the concept that the W_b_A and W_A_r matrices are pre-calculated and looked up during the loop iterations. Which will reduce the number of calculations. The implemented algorithm specified under listing G.1 does not use a lookup table for both loops. So the new memory requirements for each matrix is listed in table G.1.

So for example if we take N_el=64;X_end=276;N_z=74;N_f=150 then the total memory required is 51.5 MB. Note that it is implied that W_A_r and W_b_A are the same.

G.1.2.2 Bandwidth

The required bandwidth depends on the implementation, so here we only look at the implementation on the quad core. The algorithm is implemented in C++:

```
1 Confidential
```

Line	Bytes read	Written bytes	Number of timer per frame
7	276k	276k	74(N_z)
14	276k	276k	11100(N_z x N_f)
15	2x276k+1k	2x276k	11100(N_z x N_f)
21	2x276k	276k	11100(N_z x N_f)
28	64k+276k	276k	11100(N_z x N_f)
30	2x276k	4.4k	11100(N_z x N_f)
32	323k	323k	74(N_z)
33	323k	2.2k	74(N_z)
total	25.3 G	15.4 G	

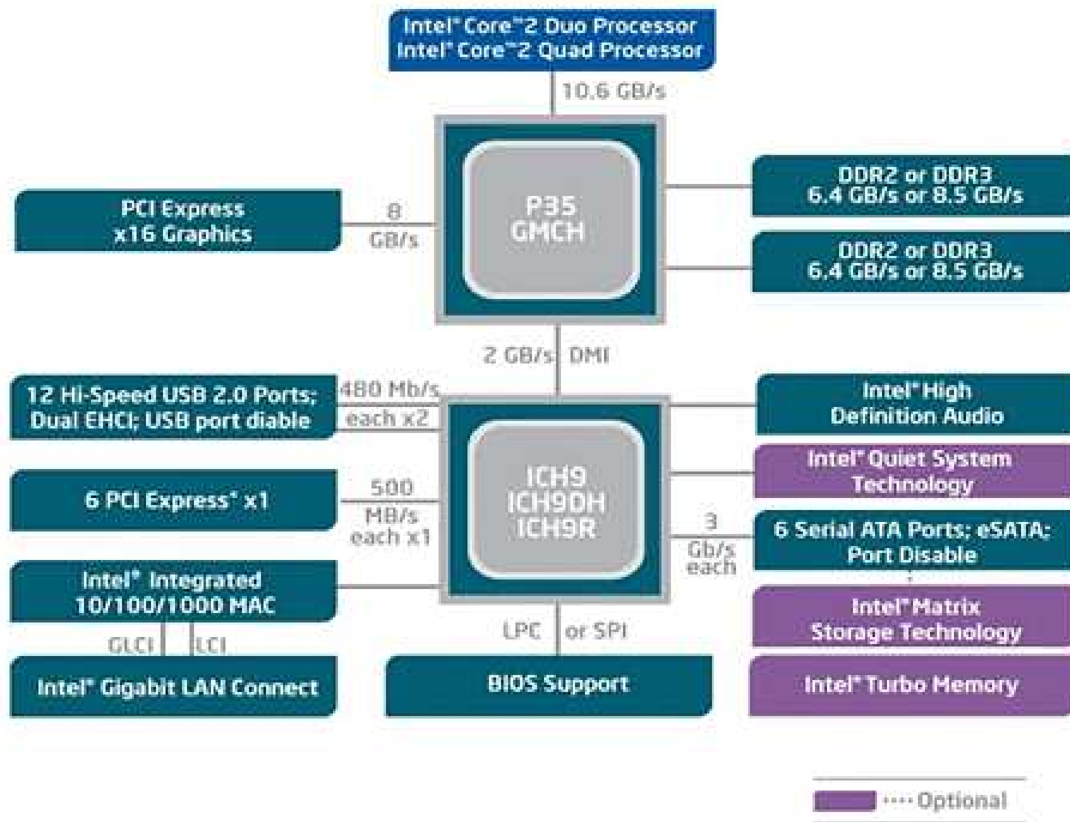
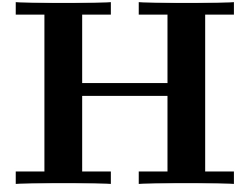


Figure G.2: Organization of the chipset [3]

The organization of the chipset used in the prototype is illustrated in figure G.2. The prototype has two DDR2 SDRAM, which both have a bandwidth of 6.4 GB/s, that would make a total of 12.8 GB/s. The processor is connected to the north bridge with a bandwidth of 10.6 GB/s, so this is the limiting factor to the memory. So if we assume that only the needed data is transferred over the north bridge, then a frame would take $\frac{25.3}{10.6} = 2.4$ seconds. Note that this can increase a lot if all data is available in the on chip cache. But it is not possible for all data to be in the cache since the cache is only 12 MB and the total data is 49.5 MB. So the cache is important but also the pre-fetchers and out of order execution. The processor is able to calculate 4 times 4 floating point operations at a frequency of 2.66 GHz what makes a rate of 42.5 GFLOPS. So the algorithm needs at least $\frac{104}{42.5} = 2.45$ seconds to calculate one frame.

Single precision versus double precision



Listing H.1: Error estimation

```
1  Error_matrix = ROI_p_p_DP-double(ROI_p_p_SP);
2
3  Column_fault = sqrt(mean(Error_matrix.^2));
4  DP_mean = sqrt(mean(ROI_p_p_DP.^2));
5  SP_mean = sqrt(mean(ROI_p_p_SP.^2));
6
7  mm = max(max(ROI_p_p_DP(:), ROI_p_p_SP(:)));
8  mse = mean(abs(ROI_p_p_SP(:)-ROI_p_p_DP(:)).^2);
9
10 psnr = 10*log10(mm.^2/mse);
11 snr = 20*log10(sqrt(mean(mean(ROI_p_p_DP.^2))) /
12                sqrt(mean(mean((ROI_p_p_DP-ROI_p_p_SP).^2))));
```

Figure H.1 illustrates the error amplitude compared to the image amplitude. The left figure illustrates the amplitude of each column. The right figure illustrates the error in each column. Note the differences in the scaling. The amplitude of the picture is in the order of 10^{14} while the error is in the order of 10^{8}. Peak Signal to Noise Ratio is a measure of quality of image operations. PSNR is calculated as in listing H.1. If we take as testing conditions:

$$N_el = 32 \tag{H.1}$$

$$N_f = 150 \tag{H.2}$$

$$zi_res = xi_res \tag{H.3}$$

In table H.1 the PSNR as well as the signal to noise ration (SNR) is compared to different pictures and resolutions. If we assume that the test cases represents all practical situations, the SNR is large enough to conclude that single precision could be an option if needed.

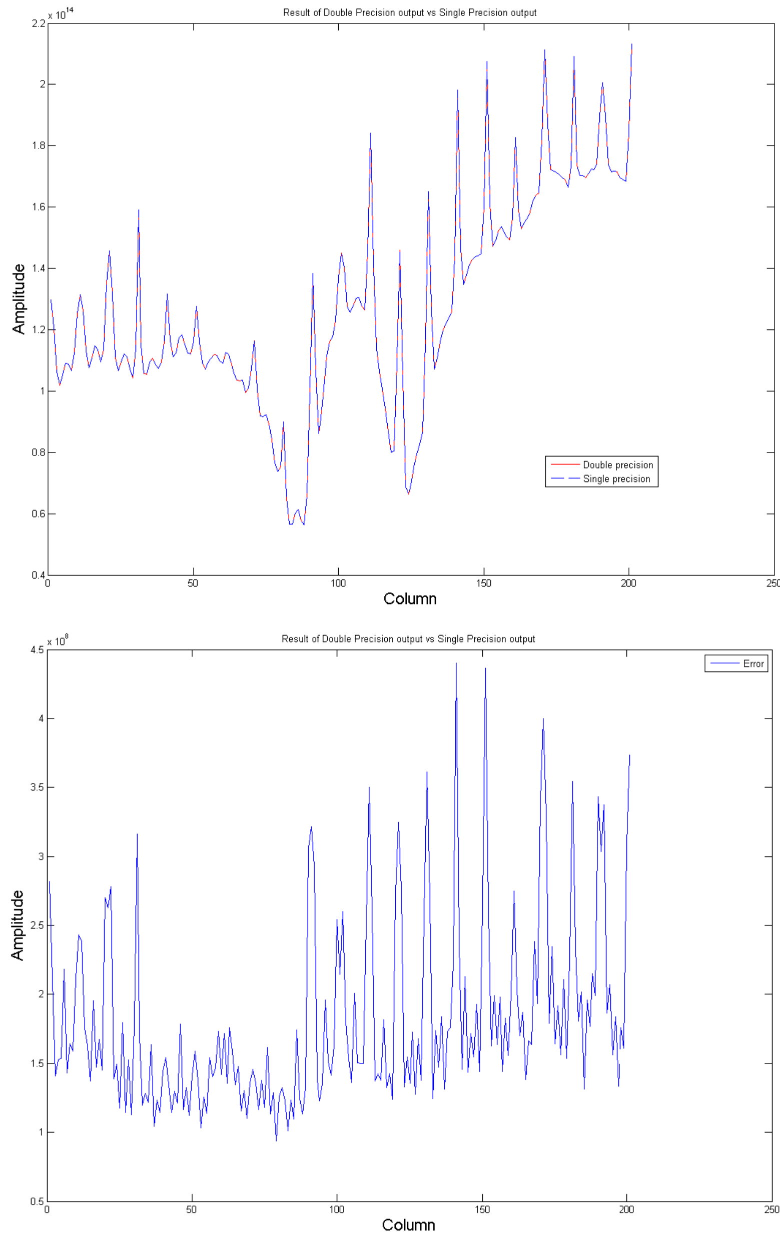


Figure H.1: Results of single precision output vs. double precision output

Data	SNR(dB) (zi_res = 1/5)	SNR(dB) (zi_res = 1/10)	PSNR(dB) (zi_res = 1/5)	PSNR(dB) (zi_res = 1/10)
LPA_probe_1mm_ slit	118.998	117.056	136.047	141.275
LPA_probe_2mm_ slit	118.144	116.584	136.307	138.656
LPA_probe_3hole_ porosity	118.305	117.471	132.883	136.255
LPA_probe_inclined_ slit	119.281	118.279	135.602	139.036
LPA_probe_inclined_ slit_lower_gain	118.749	117.935	135.452	141.906

Table H.1: (P)SNR dependent on the picture and resolution

Curriculum Vitae

A.Thomas was born on 3 January 1981 in Meppel, the Netherlands. He completed his secondary education at the Groene Hart College in Alphen a/d Rijn in 1997. From 1997 to 2001 he attended the intermediate vocational education in Leiden. In 2001 he started at the Rijswijk University of Professional Technical Education where he achieved his Bachelor of Science degree cum laude in 2004. He continued his studies in 2006 at the TU Delft computer engineering department where he is hoping to receive his Master of Science degree. Besides his education and between the degrees he worked in software and hardware design for different companies. His main interests include embedded systems and computer architecture.