Universiteit Leiden
The Netherlands

Opleiding Informatica

Hardware acceleration

of matrix multiplication

Gijsbert Maan

Supervisors:
Todor Stefanov & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

30/07/2019

Abstract

Multiplication of dense matrices is a computationally intensive task. A known way to increase the performance of this task is to perform the calculations on a GPU which utilizes parallelism. In this thesis we will determine if we can design custom hardware that is faster than current optimized GPU implementations as well as a CPU implementation. An appropriate algorithm for the multiplication was selected and implemented for CPU (baseline), FPGA board (simulated hardware acceleration board) and GPU. Then for each platform optimal settings for the algorithm were determined.

It was found that for square matrices of size 256 - 2048 the performance of the FPGA board was roughly 2 times faster over the CPU baseline and the performance of the GPU was roughly 100 times faster than the FPGA.

Contents

Chapter 1

Introduction

Matrix multiplication is an often performed task, e.g., in computer graphics, signal processing, quantum mechanics, etc. Since matrix multiplication involves many individual multiplications and additions, this process is often a bottleneck. Improvement in the speed of matrix multiplication is therefore essential. Such increase in speed can be achieved by

- reducing the number of multiplications (rather than the number of additions, because a multiplication takes more time than an addition).
- performing the calculations in parallel rather than serially, by dividing the matrix into smaller blocks.
- performing the calculations (partially) in hardware.

1.1 Problem statement

Matrix multiplication is the multiplication of two matrices A and B of size $m \times n$ and size $n \times p$, respectively, which results in a matrix C of size $m \times p$. Calculating just one element of C takes n multiplications. The complete calculation of matrix C will take $m \times p$ (number of elements of C) x n (multiplications per element). In case of square matrices, m, n and p are equal, and the total number of multiplications is n^3.
This thesis will focus on finding a kernel which can be hardware accelerated on an field-programmable gate array (FPGA), and how the size of this kernel affects the execution time on both the FPGA and CPU. We will also look at an implementation of hardware acceleration on a graphics processing unit (GPU). This gives us the following research questions:

1. What speed increase can be attained by performing dense square matrix multiplication on different hardware platforms, such as an FPGA or GPU?
2. How does the size of the kernel affect the execution time of large matrix multiplication?

3. How does the size of the matrices affect the execution time of large matrix multiplication?

1.2 Contributions

The contributions of this thesis consist of:

- A divide and conquer implementation of matrix multiplication for square matrices on a CPU.
- A high level synthesized solution for matrix multiplication on an FPGA board.
- A simple GPU implementation to comparing the CPU and FPGA.
- An evaluation of how different parameters, e.g., the kernel size and matrix size, affect the execution time on the three different hardware platforms.

1.3 Thesis Overview

This chapter contains the introduction; Chapter 2 discusses related work; Chapter 3 includes some background in order to better understand this thesis; Chapter 4 explains the matrix multiplication done on a CPU; Chapter 5 explains the matrix multiplication done on a GPU; Chapter 6 explains all the work done needed for the hardware acceleration on an FPGA; Chapter 7 evaluates the contributions and the performance of several architectures; Chapter 9 gives the final conclusion; Chapter 8 discusses the thesis and some choices made.

Chapter 2

Related Work

In the article [AIBM+19] matrix multiplication is performed on an FPGA. They do the same in the sense that their computing unit also only works on square matrices. However they use an algorithm based on circulant matrices. An advantage of their method is that they are able to perform not only the matrix multiplication operation, but also matrix-by-matrix addition, subtraction, dot product, matrix-by-vector multiplication, and matrix by scalar multiplication.

In the article [ZHSJ+09] an analysis is given between a Sparse matrix-vector multiplication on a GPU and a FPGA and their relative performances. The difference compared to my thesis, is that they use Sparse matrix-vector multiplication. They found the GPU greatly outperforming the CPU. When this article was written, in 2009, they used the Nvidia GeForce GTX 260, however due to the fast paced development in the field of GPUs, the results on a GPU from this year, 2019, might differ greatly.

In the article [CBH19] a method is given where they produced new solutions to the Laderman's problem using 23 multiplications. They do, however, try to find out if they can do the calculation with 22 instead of 23 multiplications. They differ in approach to this thesis, as they try to find a way to make matrix multiplication faster through the means of a better algorithm, by reducing the amount of multiplications, whereas this thesis focuses on a way to make the calculations faster.

Chapter 3

Background

Matrix multiplication can be performed with different algorithms. Each algorithm uses time for multiplication, addition and overhead. The simplest method is to calculate each element in succession. The advantage is that the process is easily understood. The disadvantage is that $m \times n \times p$ multiplications are needed and that, especially for large matrices, a fairly large amount of data has to be loaded into the memory for the calculation of each element of matrix C, thereby increasing the time of the processor spent on overhead. For square matrices of $n \times n$, the number of multiplications is n^3, therefore the time consumption is given as $\mathcal{O}(n^3)$. It can be easily seen that doubling the size of a matrix (to $2n$) will result in $8n^3$ calculations, an 8-fold increase.

Another way to perform matrix multiplication is by using a specialized algorithm, like the one that was developed in 2011 by Courtois, Bard and Hulme [CBH19] for the multiplication of two 3x3 matrices. The number of multiplications needed was 23 rather than 27 using the method mentioned above. This reduces the complexity to approximately $\mathcal{O}(n^{2.856})$.
Volker Strassen [Str69] developed a method for the multiplication of square matrices with size 2 where the number of multiplications was reduced from 8 to 7 at the cost of more additions, thereby achieving complexity $\mathcal{O}(n^{2.807})$.
The Divide and Conquer algorithm [CLR09] is a method in which a square matrix that has a size of any power of 2 is divided into 4 equal sub-matrices, each of which can be again divided into four parts. This can be done recursively until the matrix has a size of 2×2. Calculating many smaller matrices is a faster method than multiplying the original, larger matrices due to the fact that it is easily parallelized. Despite the cutting into smaller matrices not being a requirement for parallelization, as there are other methods, it will make later hardware acceleration on the FPGA easier. Constructing the large multiplication matrix from these smaller parts is a fairly easy process, consisting mainly of additions. Since any $m \times n$ matrix can be extended with zeros to a square $p \times p$ matrix (with p a power of 2), this method can also be applied to non-square, non-power of 2 matrices. Of all the algorithms mentioned in this paragraph, only this one is used in this thesis.
Other, more complex methods have been published, e.g., the Coppersmith-Winograd algorithm [CW90], which has a complexity of $\mathcal{O}(n^{2.375})$ followed by several relatively small improvements with the currently best

performing $\mathcal{O}(n^{2.372})$ complexity by LeGall [LG23]. These complex methods have so much overhead that they only become useful for the multiplication of very large matrices.

Chapter 4

Central Processing Unit

In order to gain a speedup with hardware acceleration, we need to determine what algorithm to use for the matrix multiplication. As discussed in Chapter 3, several options are available to choose from. In this case a divide and conquer algorithm is used.
Multiplication of two matrices can be facilitated by dividing the matrices into smaller blocks. This method is used because offering too large amounts of data to the processor will lead to degradation in performance due to cache misses.
Partitions of these matrices have constraints, i.e., not every random partitioning will work, so this thesis will be using square matrices of size 2^n, because they conform to the constraints, but also because they are well suited for a recursive algorithm. This algorithm makes it possible to create a small and simple kernel.
Another reason for choosing the divide and conquer algorithm, is that it splits the matrix in smaller blocks to perform matrix multiplication on. This means the implementation can make use of parallelism. In order to give a fair comparison to other hardware platforms, multiprocessing is used in this implementation.

Since we want to compare the hardware accelerated program with the CPU, we first need to create a C program which uses the CPU. This C program uses the divide and conquer algorithm to provide a baseline to work from. For reference see the pseudo code outlining the basics of the program. The most important functions of this program are:

1. A function, randomgen, to fill the matrix with random values within a set range (see Figure 4.1).
2. A function, muladd, to add two matrices (see Figure 4.2).
3. A recursive function, recur, splitting matrices A and B of size n into four new matrices of size $n/2$, as long as the size of the matrices are above a certain value p (the kernel size). This is a value which can be changed in the header file, and acts as a parameter which defines the size of the matrices used in the kernel for the actual multiplication. After the multiplication the recursive function also adds the resulting matrices together to reconstruct the larger matrix (see Figure 4.4).

4. A function calling the former function on 4 different threads, as there are 4 cores in the CPU used, in order to attain optimal usage of the CPU's cores (almost the exact same as the recursive function). (See Figure 4.4 and 4.5).

5. A function, mulcalc, which is called when the value p is reached. This function multiplies two matrices of size p using algorithm 1), introduced in Chapter 3 (see Figure 4.3).

The program is setup this way, because now we can use the function for the multiplication as a kernel. However, it is important to note that the size of the kernel influences the time of execution, because the smaller the kernel, the more overhead it will generate. It will also be investigated what the optimal kernel size is for a chosen matrix size n.
With this program it is possible to measure the amount of time spent in the overhead and the kernel, where the actual matrix multiplication is executed, by inserting timers at strategic points. For the actual calculations the timers were removed. This is critical to determine if the amount of time spent in the kernel is large enough for an effective acceleration.

Figure 4.1: A function to fill a matrix with random numbers within a set range.

```
void randomGen(int size, float** Z){
        for(int i = 0; i < size; i++){
                for(int j = 0; j < size; j++){
                        Z[i][j] = randomNumber();
                }
        }
}
```

Figure 4.2: A function to add two matrices.

```
void mulAdd(int size, float** A, float** B, float** C){
        for(int i = 0; i < size; i++){
                for(int j = 0; j < size; j++){
                        C[i][j] = A[i][j] + B[i][j];
                }
        }
}
```

Figure 4.3: A function to multiply two matrices.

```
void mulCalc(float** A, float** B, float** C){
        for(int i = 0; i < kernalSize; i++){
                for(int j = 0; j < kernalSize; j++){
                        C[i][j] = 0;
                        for(int k = 0; k < kernalSize; k++){
                                C[i][j] += A[i][k] * B[k][j];
                        }
                }
        }
}
```

Figure 4.4: A recursive function splitting and recreating matrices.

```
void recur(int size, float** A, float** B, float** C){
        if (size > kernalSize){
                int hsize = size / 2;
                for(int i = 0; i < hsize; i++){
                        for(int j = 0; j < hsize; j++){
                                A1[i][j] = A[i][j];
                                A2[i][j] = A[i][hsize + j];
                                A3[i][j] = A[hsize + i][j];
                                A4[i][j] = A[hsize + i][hsize + j];

                                B1[i][j] = B[i][j];
                                B2[i][j] = B[i][hsize + j];
                                B3[i][j] = B[hsize + i][j];
                                B4[i][j] = B[hsize + i][hsize + j];
                        }
                }
                (When starting threads, Figure 4.5 goes here)
                {recur(hsize, A1, B1, D1);
                recur(hsize, A2, B3, D2);
                muladd(hsize, D1, D2, C1);
                recur(hsize, A1, B2, D1);
                recur(hsize, A2, B4, D2);
                muladd(hsize, D1, D2, C2);
                recur(hsize, A3, B1, D1);
                recur(hsize, A4, B3, D2);
                muladd(hsize, D1, D2, C3);
                recur(hsize, A3, B2, D1);
                recur(hsize, A4, B4, D2);
                muladd(hsize, D1, D2, C4);}

                for(int i = 0; i < hsize; i++){
                        for(int j = 0; j < hsize; j++){
                                C[i][j] = C1[i][j];
                                C[i][hsize + j] = C2[i][j];
                                C[hsize + i][j] = C3[i][j];
                                C[hsize + i][hsize + j] = C4[i][j];
                        }
                }
        }else{
                mulCalc(A, B, C);
        }
}
```

Figure 4.5: Calling the 4 threads for parallelization.

```
pthread_create(recur(hsize, A1, B1, D1));
pthread_create(recur(hsize, A2, B3, D2));
pthread_create(recur(hsize, A1, B2, D3));
pthread_create(recur(hsize, A2, B4, D4));
joinThreads();
muladd(hsize, D1, D2, C1);
muladd(hsize, D3, D4, C2);

pthread_create(recur(hsize, A3, B1, D1));
pthread_create(recur(hsize, A4, B3, D2));
pthread_create(recur(hsize, A3, B2, D3));
pthread_create(recur(hsize, A4, B4, D4));
joinThreads();
muladd(hsize, D1, D2, C3);
muladd(hsize, D3, D4, C4);
```

Chapter 5

Graphics Processing Unit

Almost all computers nowadays contain a graphics processing unit (GPU), either a dedicated or embedded on the motherboard. In everyday use these devices manipulate and output images to a target output device, like a monitor. GPUs are also a key component in computer gaming, making it possible to render high quality images in real time, as well as speeding up video rendering in general.
GPUs are highly useful because of their parallel structure. A GPU contains thousands of simple cores, where each core can be used to calculate a chunk of data. This property is very useful for Matrix multiplication, as it is possible to compute many blocks of a subdivided matrix at a given time.

The GPU used in this thesis is the Nvidia GeForce GTX 1070, a consumer rated graphics card, containing 1920 [Nvi] CUDA cores, and running at 1683 MHz. To perform matrix multiplication on the GPU, the cuBLAS library, which comes default with the CUDA installation, is used. This is a hand-tuned library, known to achieve good performance on Nvidia GPUs. The results of the execution time on the GPU will be used to give an indication of how close the performance of the FPGA and CPU implementations will be, compared to a known way of fast matrix multiplication.
In Figure 5.1 Pseudo code is given to give an overview of the program. The main differences between the CPU and GPU code is that memory has to be allocated for the GPU, and there are no recursive function calls, as the cublaSgemm function takes care of the matrix multiplication and parallelization. The Host allocation and memcpy (not shown) calls for A and B seem redundant, however they are needed for measurements and are used to mimic the transfer matrices from the host to the GPU.

Figure 5.1: Pseudo code of the GPU program.

```
void mulCalc(cublasHandle_t &handle, const float *A, const float *B, float *C,
             const int size){
        const float alf = 1;
        const float bet = 0;
        const float *alpha = &alf;
        const float *beta = &bet;

        cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, size, size, size, alpha, A,
                    size, B, size, beta, C, size);
}

int main(){
        int size = 8;
        cublasHandle_t handle;
        cublasCreate(&handle);

        float *A = malloc(size * size);
        float *B = malloc(size * size);
        float *C = malloc(size * size);

        float *AGpu, *BGpu, *CGpu;
        cudaMalloc(&AGpu, size * size);
        cudaMalloc(&BGpu, size * size);
        cudaMalloc(&CGpu, size * size);

        mulRandGen(AGpu, size);
        mulRandGen(BGpu, size);

        mulCalc(handle, AGpu, BGpu, CGpu, size);

        cudaMemcpy(C, CGpu, size * size, cudaMemcpyDeviceToHost);
        return 0;
}
```

Chapter 6

Custom Hardware

Hardware acceleration is done by picking a code segment in which the program resides for most of the runtime, and optimize this to run significantly faster on a different hardware platform. It is critical that this code segment is chosen correctly and the program resides in this code segment. for at least 80% of the runtime, otherwise gaining a speedup will be significantly harder.

6.1 FPGA

In this thesis the hardware acceleration is done by simulating a Field-Programmable Gate Array. This is a platform which consists of programmable logic blocks, which can be routed in different ways, making many different configurations possible. These FPGAs are generally programmed by writing hardware description language (HDL) code, such as VHDL [Nav97] or Verilog [TM08]. This thesis, however, uses High Level Synthesis (HLS), to generate HDL code. When using HLS, instead of writing in a HDL language, a higher level programming language can be used, such as C.
The program used for HLS is Vivado HLS 2018.2 [Xilc]. This program has several functions, like C synthesis and C/RTL co-simulation. The former, C synthesis, is the process of converting the written C code into an RTL level code, like VHDL or Verilog. C synthesis will only be used on the kernel, because the kernel will be the only part of the hardware acceleration. C/RTL co-simulation uses a test bench to verify the functionality of the kernel.

As said before, C can be used to describe the kernel, however this is not entirely true. The kernel will be implemented as hardware, therefore the size of types like arrays have to be declared when used in a function, because the hardware can not be changed once it has been set (see Figure 6.1).

Figure 6.1: The modified version of the C kernel

```
#include "matrixmul.h"

void mulCalc(float A[kernalSize][kernalSize], float B[kernalSize][kernalSize],
float C[kernalSize][kernalSize]){
        for(int i = 0; i < kernalSize; i++){
                for(int j = 0; j < kernalSize; j++){
                #pragma HLS UNROLL
                        C[i][j] = 0;
                        for(int k = 0; k < kernalSize; k++){
                                #pragma HLS UNROLL
                                C[i][j] += A[i][k] * B[k][j];
                        }
                }
        }
}
```

In order to make the kernel more efficient, loop unrolling is applied by adding pragma HLS UNROLL [Xila]. This transforms loops by creating multiple copies of the loop body in the RTL design, thereby making it possible to execute the statements in a loop in parallel, as the statements are independent from each other. This increases the total size of the kernel and increases the performance.

The kernel can now be synthesized for a chosen FPGA board, and statistics like resources used are generated. We can now see the time it takes to execute the kernel, as well as all the hardware parts used per the kernel. It is also possible to put more than one of these kernels on the FPGA board, thereby creating parallel computations. The limitation of the amount of these kernels is the size of the board. In this case 50 independent kernels are put on the board, which would lead to about a 50 times speedup in the kernel execution time. A host CPU has to be used to transfer the data to the board RAM, time it would take for a host CPU copy data to and from the board RAM can be calculated because we know the bandwidth of the board. The simulation is done by calculating times, therefore there is no actual program moving data or directing the blocks. The overhead combined with the execution time of the kernels and the time it takes to move the data to the board RAM is the final number used for the evaluation.

6.2 Choosing an FPGA

In order to perform the hardware acceleration, a hardware board has to be chosen.
To determine the best board, we look at logic and memory resources, as well as DSP slices. More logic gates, memory blocks and DSP slices, means it is possible to execute more kernels in parallel.
Besides looking at the resources, we also have to consider the licenses of these boards.

In this thesis several boards were considered, such as the Kintex UltraScale KU095, KU115 or Virtex Ultrascale XCVU190 [Xilb] (see Table 6.1). The Virtex board contains more logic and memory resources than the Kintex boards, however they lack in DSP slices. Between the Kintex boards, the KU115 has more resources. After C synthesis, the amount of resources needed for the kernel are known. In this case the KU115 came out as the best option for this thesis, since DSP slices were the limiting factor. The board runs at 100 MHz.

Table 6.1: Table with considered FPGAs

Name	Total board RAM (Mb)	DSP Slices	Flip Flops	LUTs
Kintex UltraScale KU095	59.1	768	1,075,200	537,600
Kintex UltraScale KU115	75.9	5,520	1,326,720	663,360
Virtex Ultrascale XCVU190	132.9	1,800	2,148,480	1,074,240

Chapter 7

Evaluation

As seen in Chapter 4, our program has several parameters, such as the size of the kernel or the size of the matrices to be multiplied. In this chapter several experiments will be described in order to evaluate the performance of our algorithm and the differences between the CPU, FPGA, and GPU implementations of our algorithm. All measurements are done under the same conditions, i.e., no other programs run on the hardware platform.

The measurements of the CPU were done with timers. The total time does not include the time to allocate the initial matrix or the filling of this matrix with random numbers, but does contain the recursive calls and their matrix memory allocation, as well as the time spent in the kernel. The measurements for the FPGA are the execution time of the kernels combined with the timed overhead from the initial C program, which was used for the CPU as well, and the calculated transfer time from the CPU to the board RAM. This gives us the following formula; $executiontime = CPUoverhead + (timetakenfor1kernel \times amountofkernelcalls)/amountofkernelsontheboard + transfertime$. Finally the measurements for the GPU were done with timers. It does not contain the allocation for the CPU, filling of the matrices or the creation of the handle to the cuBLAS library, but does contain the allocation for the GPU, the bus transfer time and the execution time of the library function.

Table 7.1 shows specifications of the Hardware used.

Table 7.1: Hardware specifications

Name	Clock speed (MHz)	amount of cores
Intel core I5 7600K	4600	4
Nvidia GeForce GTX 1070	1683	1920
Kintex UltraScale KU115	100	-

7.1 CPU Overhead

As stated in Chapter 4, to get a speedup, by accelerating the kernel, the time spent in the kernel should be at least 80% of the total runtime. Figure 7.1 shows the time spent in the kernel in green versus the overhead in purple in seconds. This experiment is the only one in this thesis not using multiprocessing, because it was used to find the relative difference in the execution time between the overhead and the kernel, and see if relatively enough time is spend in the kernel.

Figure 7.1: Time spent in the kernel versus time spent in the overhead

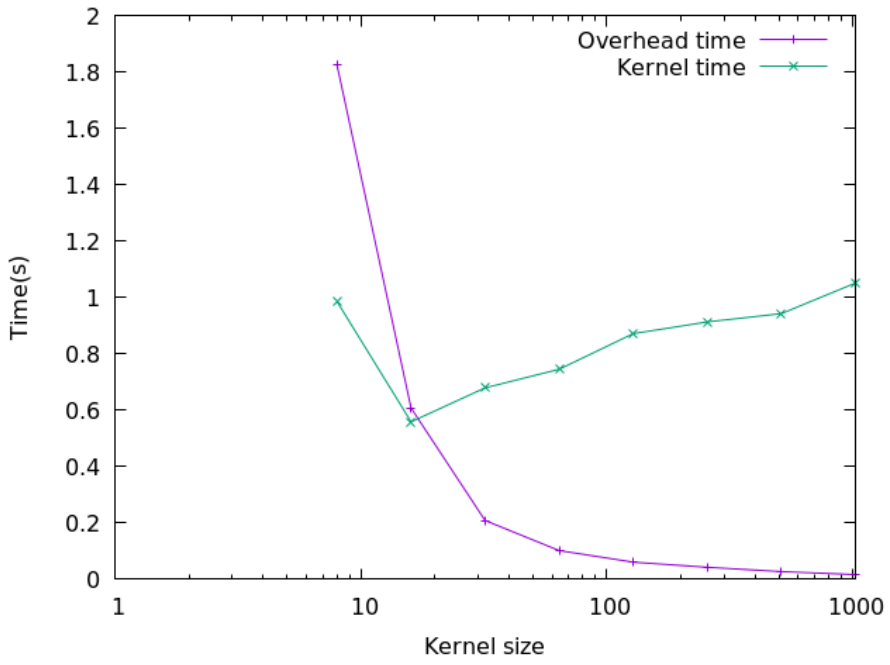

Figure 7.1 shows the amount of overhead is large when using a small kernel, and keeps decreasing with increasing the kernel size. This is because smaller kernels require more recursive calls and thus increase the overhead. The figure also shows that the kernel time increases with a larger kernel size. The most probable explanation for this is that, as kernel size increases, matrices processed by the kernel no longer fit L1 cache causing the kernel time to increase.

7.2 CPU average execution time

In order to determine the execution time of a matrix multiplication of matrices A and B, both of them are filled with random floating point numbers between 0 and 50. To make sure the randomness of the matrices have no significant effect on the outcome of the execution time, the calculation is done 1000, 500, 250 and 125 times and for different sizes of matrices. Average runtimes for single kernel execution and standard deviations were calculated and are given in Table 7.2

Table 7.2: Time used for the multiplication of square matrices of size 128, 256 and 512 with a kernel size of 64, values are (average $\pm$ standard deviation) in milliseconds.

Matrix size \ Repeat count	1000	500	250	125
128	0.327 ± 0.112	0.306 ± 0.1050	0.341 ± 0.489	0.376 ± 0.447
256	5.520 ± 0.762	5.490 ± 0.773	5.508 ± 1.039	5.545 ± 0.834
512	35.433 ± 3.909	34.987 ± 3.448	35.715 ± 3.741	35.226 ± 3.734
1024	242.240 ± 2.693	242.184 ± 1.146	242.290 ± 2.265	242.323 ± 0.817

The table shows that the average multiplication time for a given matrix size is independent on the number of runs. The standard deviation is small for larger matrices. The reason these standard deviations are increasing with smaller matrices, are presumably because small interruptions have a big effect in small matrices, since the execution time is quite short already. To further look into why standard deviation is what it is, we multiplied 512×512 matrices 125 and 1000 times.

Figure 7.2: Frequency of execution time and outliers

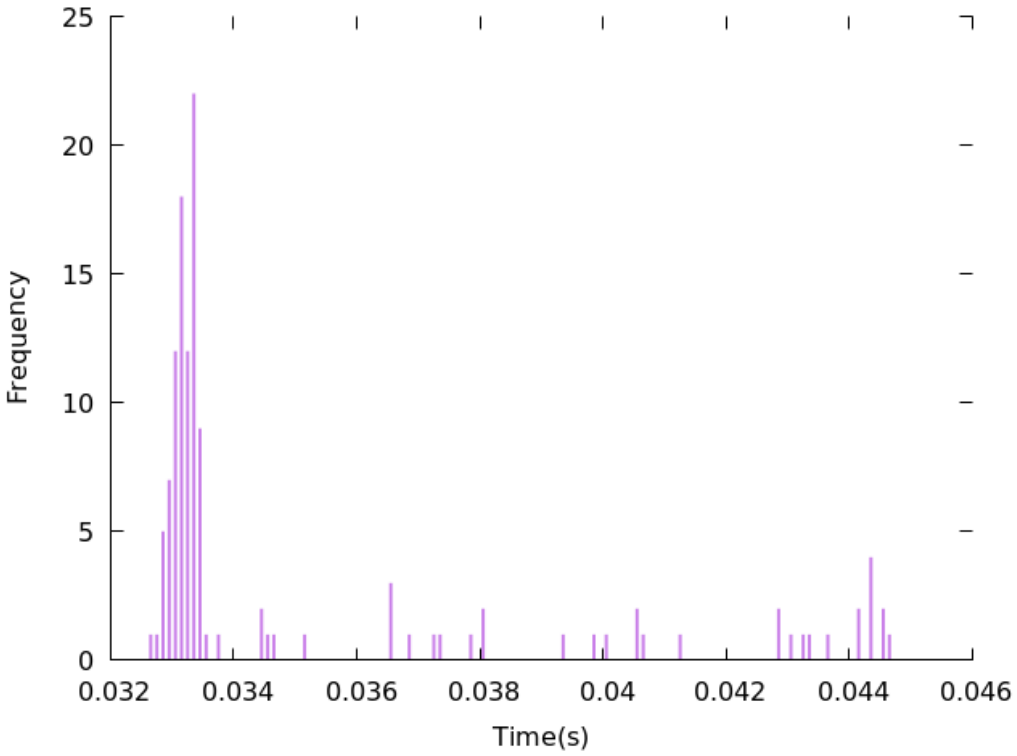

Figure 7.3: Frequency of execution time and outliers

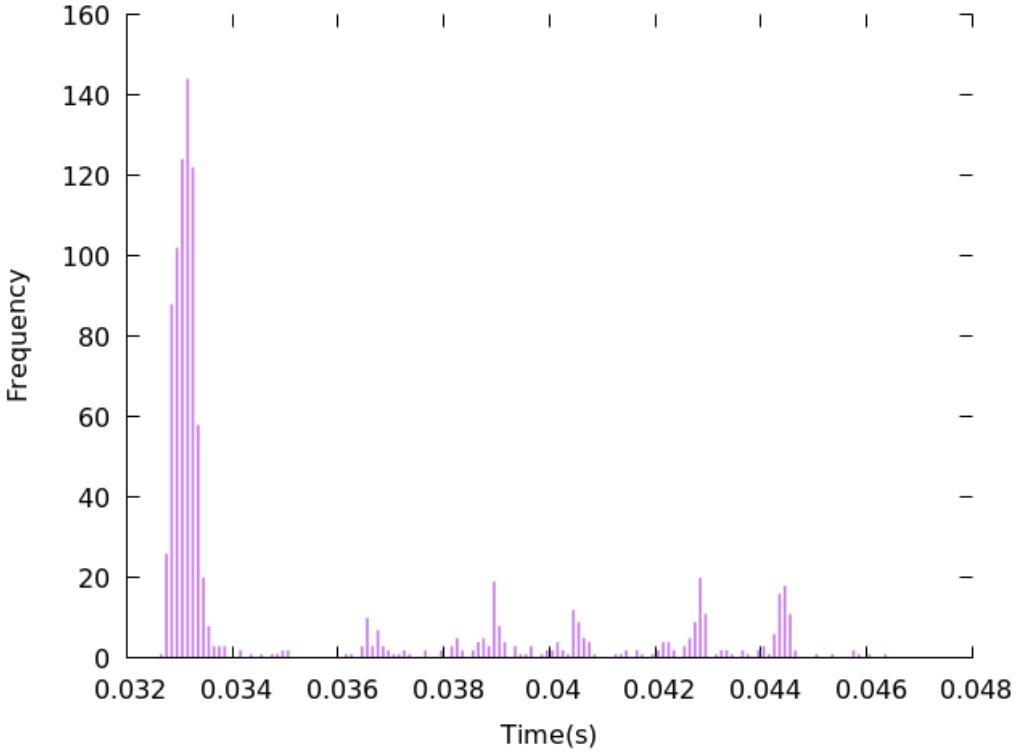

Figure 7.2 and 7.3 both represent matrix multiplications of 512×512 matrices. The time taken to execute one multiplication of matrices $A \times B$ is plotted on the x-axis, the frequency of the execution time is given on the y-axis. Figure 7.2 shows 125 multiplications of $A \times B$, while Figure 7.3 shows 1000 multiplications.

Figure 7.2 shows around 30 outliers above 0.034, while Figure 7.3 shows around 200 outliers above 0.117 seconds. The outliers are on every occasion higher than the average, as mentioned above, indicating that sometimes execution took longer. There also seems a pattern of several high peaks of outliers, this might be caused by CPU interruptions.

7.3 Kernel sizes effect on execution time

In this experiment, we look at the difference between the performance of the CPU and the FPGA. Since the standard deviation was very low and the average calculation time was fairly independent, we feel confident that using the average time from 100 calculations is more than adequate for comparison. We determine the influence of the kernel size on the calculation time, as can be seen in Figure 7.4 (the time used against the size of the matrix in the kernel).
An example of the calculation of the FPGA execution time for 50 kernels of size 64 is the following: 64(size)$\times$64(size)$\times$2(amount of matrices)$*$4(size of float)$*$50(amount of kernels) $= 1,638,400$ bytes or 1.6 MB. The speed of the board RAM is 2400 MB/s, giving us a transfer time of $1.6/2400 = 0.00066$ seconds.

When multiplying 1024×1024 matrices, the kernel has to be called 4096 times, meaning that this data transfer has to be done 82 times in total, giving us $82 \times 0.00066 = 0.05466$ seconds of total transfer time.
Now we determine the total execution time for a 1024×1024 matrix with size 64;
execution time $= 0.0315271(\text{overhead}) + (1380838.979 \times 10^{-9} * 4096)/50)(\text{kernel time}) + 0.05466(\text{transfer time}) =$ 0.199 seconds.
All of these calculations are done with two 1024 by 1024 matrices. The vertical axis shows the time in seconds. The horizontal axis indicates the size of the matrix used in the kernel.

Figure 7.4: Matrix multiplication of a 1024×1024 matrix with different kernel sizes.

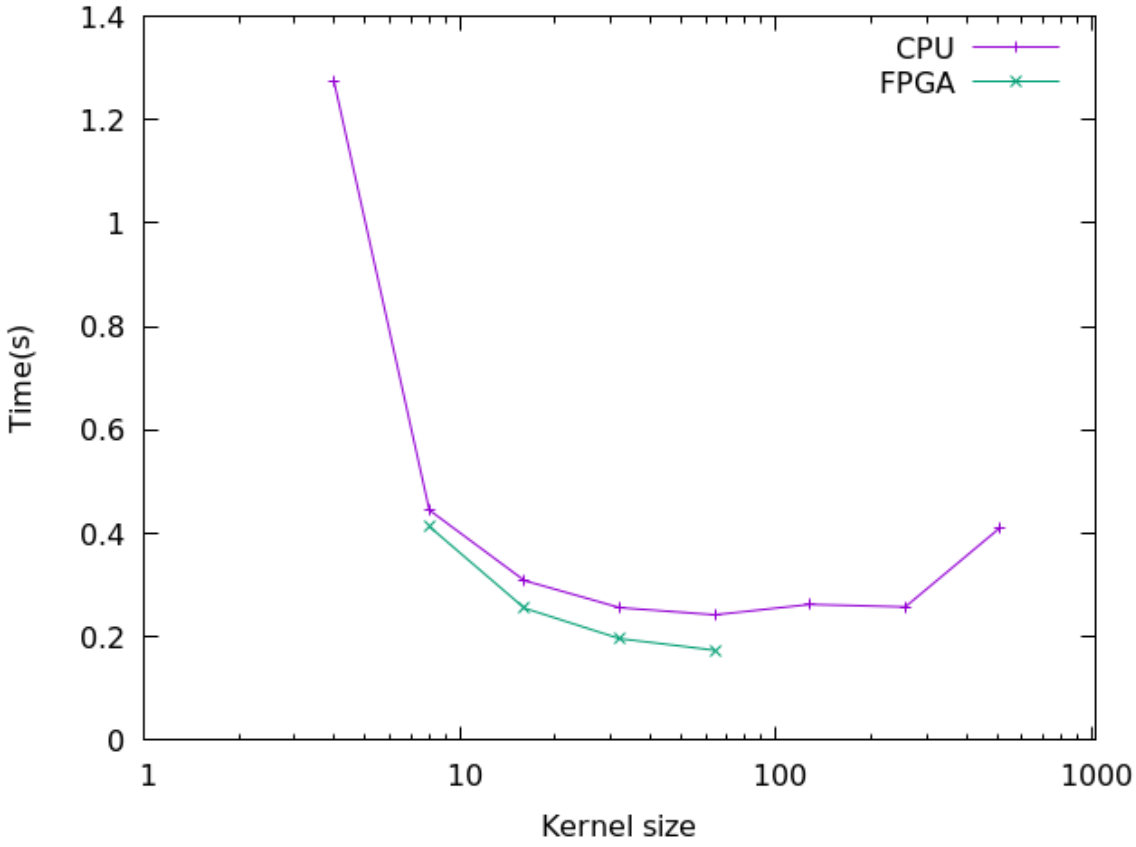

Figure 7.4 shows two lines, a purple one which represents the CPU performance and a green one which represents the FPGA performance. The kernel size of the FPGA has a maximum of 64, because of size limitations of the synthesis program. The figure shows that the FPGA performs significantly better when the kernel size increases, presumably because there will be less overhead. The CPU shows a similar trend, however, at around the matrix kernel size of 64 the figure starts to go up again, presumably because the matrices are too big for the first-level cache, causing a reduction in performance.
When we compare the CPU and the FPGA we can see there is a slight increase in performance when using an FPGA over a CPU, when using a kernel of size 32 or 64.

7.4 Matrix sizes effect on execution time

In this experiment, we look at the difference between the performance of the matrix multiplication on the CPU, FPGA, and GPU. To determine the time spent for the matrix multiplication, the matrices are filled with random floating point numbers between 0 and 50, and the program is ran to determine the execution time. This process is repeated 100 times for all the different size matrices and the average of these results are taken. A kernel size of 64 is chosen for both the CPU and FPGA, as this is the optimum for both methods, determined by the experiment in Section 7.3. The GPU does not make use of a kernel, as it uses a BLAS kernel, which does not expose the used parallelization method. Figure 7.5 shows the time in milliseconds on a log scale versus the size of the two matrices both $n \times n$.

Figure 7.5: Matrix multiplication on the CPU, FPGA, and GPU with different matrix sizes.

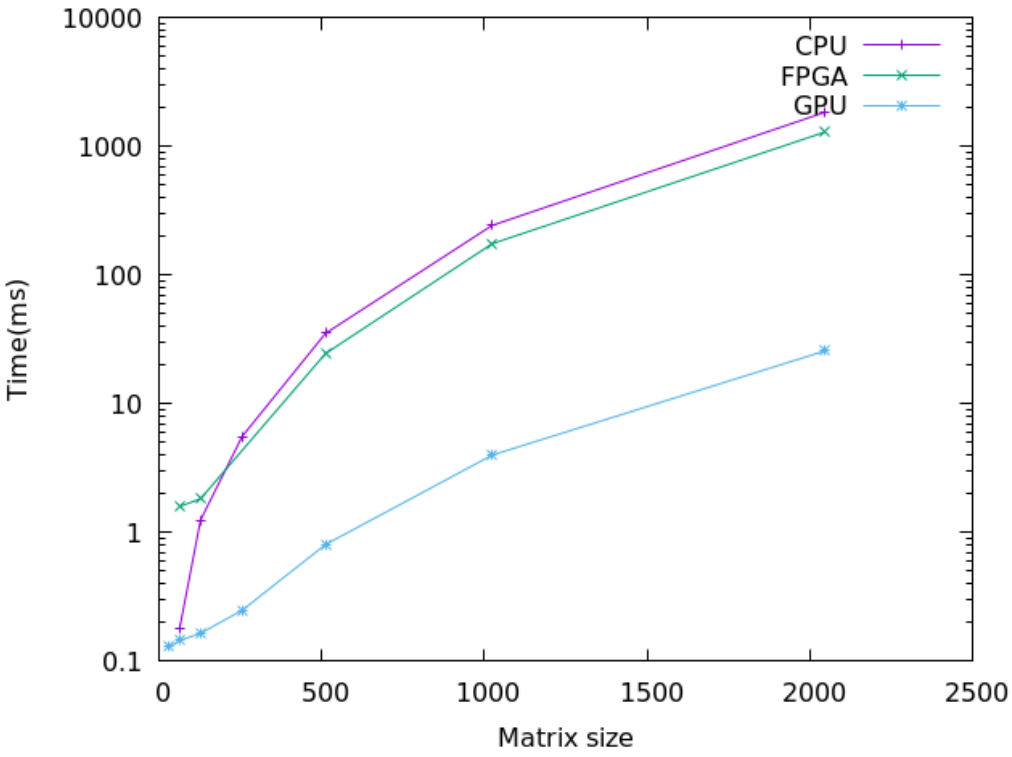

Figure 7.5 shows three lines, the green line for the FPGA, the purple line for the CPU and the blue line for the GPU. The figure shows that the GPU performs better compared to the FPGA and CPU implementation. Presumably this is because the GPU uses significantly more parallelization compared to the FPGA and CPU. Since the matrix multiplication contains a lot of parallelizable operations, the GPU performs best. The GPU can utilize 1920 cores, and the FPGA only 50 kernels, meaning the GPU has an advantage of about 40 times as much parallelization. However, the FPGA implementation shows a better execution time compared to the CPU, because it uses more parallelization compared to the CPU and has a dedicated kernel. The CPU and FPGA lines cross at a low matrix size, presumably because the overhead on a FPGA is greater when calculating

smaller matrices, as well as the FPGA not being able to utilize parallelization as much.
The frequency at which the hardware runs is not the same for each hardware platform. The CPU has the highest frequency of 4600 MHz followed by the GPU with 1683 MHz and the FPGA which runs at only 100 MHz. The FPGA cannot run at a much higher frequency due to the fact that it has a lot of hardware configurability. When matching the frequency, the FPGA would perform significantly closer to the GPU. Another thing to note is that the CPU implementation is not optimal and its performance can be further improved through hand-optimization and by employing vectorization. This would likely result in the CPU outperforming the FPGA, however the FPGA implementation also is not at its optimal level, due to being on an FPGA board, but is already competitive with the CPU.

Chapter 8

Discussion

This thesis uses a straightforward algorithm to calculate the kernels and not a more advanced one like the Strassen algorithm. The reason is, because this thesis is not meant to find or use a better algorithm, but use hardware acceleration to create a speedup. Besides, we wanted to create different size kernels, which is not possible when using algorithms like the Strassen algorithm.
The FPGA that was used, ran at 100 MHz while the other hardware platforms (CPU and GPU) had a much higher frequency. It can be argued it is unfair to compare these methods due to the difference in the operating frequency, however, it is possible to create a chip with the configuration of the FPGA, which would be able to run at a higher frequency than the current FPGA does.
Besides the difference in frequency, there is also a difference in the power consumption. A GPU uses several hundred watts while a CPU usually uses a bit less than a hundred. The FPGA does, however, use less due to the lower frequency it runs at. For different applications it might be good to look at what is more important , a lower power usage, or a lower execution time. For future research it can be looked into which of the CPU, GPU or FPGA performs best for these different circumstances i.e., to explore and compare them based on the achievable performance per watt.

Chapter 9

Conclusions

Matrix multiplication is a basic operation, that can significantly be improved through several methods, such as using better algorithms, hardware acceleration, or parallelizing the process of the multiplication. The last one being the easiest to implement with a significant performance gain. The results showed that when using the divide and conquer method, presented in this thesis, it can cause a lot of overhead when using smaller kernels, which results in a longer execution time for both the CPU and the FPGA.
When using the FPGA for hardware acceleration we saw that by using parallelization it results in a slightly better performance over the CPU, i.e., the method which uses the least amount of parallelization. However, when comparing the results of the FPGA to the implementation on the GPU, it shows that the GPU is by far superior due to the possibility of utilizing parallelization to a much greater extend.

Bibliography

[AIBM+19] Asgar Abbaszadeh, Taras Iakymchuk, Manuel Bataller-Mompen, Jose V. Francs-Villora, and Alfredo Rosado. Anscalable matrix computing unit architecture for fpga,and scumo user design interface. *Electronics*, 8:94, 01 2019.

[CBH19] Nicolas T. Courtois, Gregory V. Bard, and Daniel Hulme. A new general-purpose method to multiply 3x3 matrices using only 23 multiplications. *arXiv:1108.2830 [cs.SC]*, 2011-08-19.

[CLR09] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[CW90] Don Coppersmith and shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9:251–280, 1990.

[LG23] Franois Le Gall. Powers of tensors and fast matrix multiplication. *ISSAC '14*, pages 296–303, 2014-07-23.

[Nav97] Zainalabedin Navabi. *VHDL: Analysis and Modeling of Digital Systems*. McGraw-Hill, 2 edition, 1997.

[Nvi] Nvidia. Geforce gtx 1070 specification sheet. `https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1070/specifications`. accessed; 2019-06-20.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[TM08] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Springer, 5 edition, 2008.

[Xila] Xilinx. pragma hls unroll. `https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uyd1504034366571.html`. accessed; 2019-07-25.

[Xilb] Xilinx. Ultrascale fpga prodcut tables and prodcut selection guide. `https://www.xilinx.com/support/documentation/selection-guides/ultrascale-fpga-product-selection-guide.pdf`. accessed; 2019-06-20.

[Xilc] Xilinx. Vivado high-level synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html#trainingSupport`. accessed; 2019-07-25.

[ZHSJ+09] Yan Zhang, Yasser H. Shalabi, Rishabh Jain, Krishna K. Nagar, and Jason D. Bakos. Fpga vs gpu for spare matrix vector multiply. *IEEE Explore*, pages 255–262, 12 2009.