Embedded Selforganizing Systems

Issue Topic: "Hardware - Accelerators"

# Comparison of acceleration methods of matrix calculations in embedded systems

Johannes Götze
Technische Universität Chemnitz
Computer Engineering
johannes.goetze@s2015.tu-chemnitz.de

Rene Schmidt
Technische Universität Chemnitz
Computer Engineering
rene.schmidt@informatik.tu-chemnitz.de

Wolfram Hardt
Technische Universität Chemnitz
Computer Engineering
wolfram.hardt@informatik.tu-chemnitz.de

**Abstract[1]—In today's algorithms for sound localization techniques, matrix calculations are ubiquitous. Therefore, this work deals with the analysis of matrix calculations and their possible realization on embedded systems. For this purpose, common acceleration technologies such as processors, GPU processing and acceleration with the help of FPGAs are compared. The results show that a graphics chip is capable to accelerate such a matrix vector multiplication compared to an implementation on a processor. Therefore a runtime of an implementation on an FPGA cannot be achieved by a GPU**

***Keywords—hardware acceleration, matrix calculations, Cuda, OpenCL, embedded systems***

## I. Introduction

Matrices are often used in computer science, for example as a storage structure for graphs or a matrix of coefficients for a filter. In a variety of application scenarios, the calculation with matrices is required. For example in computer graphics: Matrices are used to perform coordinate transformations [1]. In the field of optics, transfer matrices are used to analyse the alteration of light rays by optical components [2].

If very large matrices are multiplied with each other, an enormous computational power is required, because of the complexity of $O(N^2)$ of such a multiplication. So it can come to the fact that in a complex system, the matrix calculation becomes a bottleneck and then this needs to be accelerated. The trivial approach would be to use a more powerful processor.

To apply this approach in embedded systems is not the best way, because of the restriction of power consumption, available space and weight of such a system. With an increasing number of matrix calculations, the computing power of an embedded processor is often not sufficient. For this reason, other hardware acceleration methods are required. These acceleration methods can be specialized to the problem and thus generate a higher data throughput.

Already for a long time quite popular and used as hardware accelerator, FPGAs are predestined for such a field of application. They are energy efficient and flexibly adaptable. Another acceleration technology, not yet used quiet often, is to program a graphics chip in the field of embedded systems.

Graphics cards are and were mainly used for graphics applications. They are mostly used to display computer generated images on the screen. But by developing architectures and libraries, the graphics chip can also be used to accelerate applications that are not specialized in graphical operations. This makes them more flexible and relatively easy to use by the programmer.

In fact that there are different acceleration methods and also different number formats, in which the matrix elements can be, it is the goal to find out which variant is used in which application area on the best. The runtimes are to be examined depending on the one hand on the number formats, but also the problem size. So in which respect the running times in comparison to the matrix size and number of calculations. In comparison are different variants on a GPU, one FPGA implementation and one variant on an embedded processor.

## II. State of the Art

### *A. FPGAs as a hardware acceleration method*

FPGAs are used as an established method when it comes to hardware acceleration. An FPGA (Field Programmable Gate Array) offers the possibility to implement algorithms using hardware description languages like VHDL. This offers the advantage of great adaptability and versatility.

[1] Copyright © 2019 by ESS Journal

An FPGA generally consists of logic blocks which are connected to each other and to the I/O blocks. The switch matrices allow to change these connections. The logic blocks are again composed of several lookup tables and flipflops connected behind them. With the lookup tables it is possible to program logical operations. Thereby a truth table is realized. In such a table an output is defined for each variant of the bit inputs. Technically this is realized with several interconnected multiplexers. With four inputs of a lookup table it is possible to implement all boolean operations with four inputs (e.g. AND, OR...). Usually such lookup tables have four to six inputs. The output is then buffered in flip-flops if necessary [3].

### *B. An FPGA as an accelerator for a matrix vector calculation*

Matrix multiplications are often required for sound localization procedures. The algorithms need the matrices to calculate coefficients of the methods with actual sensor values. Beamforming is one such method. This algorithm is used for position detection to improve the quality of signals. Since this algorithm with a matrix vector multiplication requires a high computational effort, the implementation is a challenge for the world of embedded systems.

The algorithm implemented in [4] uses complex numbers. The calculation consists of two parts. In the first step (see equation 1), a 1 x N vector is multiplied by an N x N matrix. The obtained 1 x N vector must now be multiplied by an N x 1 vector in the second part (see equation 2). The size of the vectors and the matrix represents the number of sensors used for localization. In the following examples, N = 3, i.e. 3 sensors, is selected.

The first part requires nine complex multiplications and six complex additions. With the three multiplications and two additions in the second part of the calculation, a total of twelve multiplications and eight additions of complex numbers are required. In general $N^2 + N$ complex multiplications and $N^2 - 1$ complex additions for the calculation are required. In order to guarantee a fast and efficient calculation, the calculation was

$$\begin{pmatrix} v_1 & v_2 & v_3 \end{pmatrix} \cdot \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} \quad (1)$$

$$\begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = z \quad (2)$$

implemented on a Xilinx Zynq SoC Board. This board combines an FPGA with an ARM processor and uses the standardized AXI interface for communication. The FPGA was selected because of its ability to parallelize and thus to handle the matrix vector multiplication.

The calculation with N = 3 is divided into three parallel vector multiplications. These are performed independently of each other and their results are summed up at the end. In the design (see Figure 1), just as in the example for a vector size of N=3, the three parallel strands are easy to recognize. In the first step, each complex multiplier calculates three complex products in order to sum them up in the second step. In the third step, each multiplier calculates its part of the vector product. In the last step, the final result of the matrix vector multiplication can be calculated by summing up. The same number of complex multipliers are required for a certain number of sensors in the FPGA design.

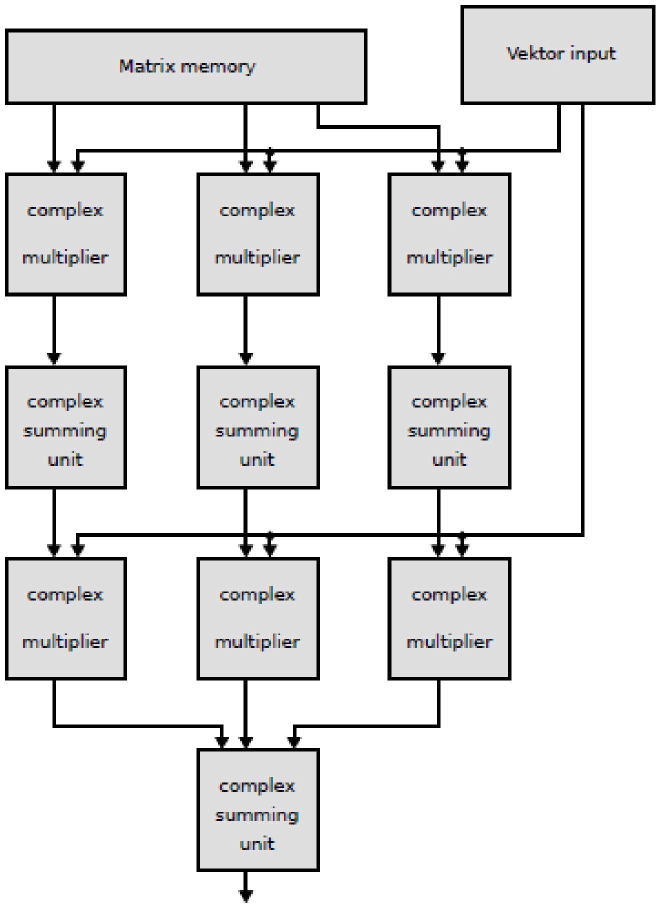

Fig. 1. Simplified FPGA-Design [4]

Due to the parallelization, the number of sensors used (and thus the size of the matrices) has no significant influence on the runtime, since correspondingly more hardware resources are used. Compared to an implementation on an ARM core, which is also available on the chip, a considerable speedup was achieved. This required an average of 500 ms for the calculation at a clock frequency of 700 MHz. The FPGA design, on the other hand, requires only 4 ms at a frequency of 100 MHz. So the FPGA is much faster but also more efficient because it consumes less power at lower clock speeds. It can utilize each clock very efficiently and thus increase data throughput. Please note that in the actual design the calculation is not yet finished after the scalar product. In further steps, the absolute value is calculated and a maximum search is performed. However, these parts only have an influence of less than one microsecond and can therefore be neglected in comparison.

### *C. GPUs as a hardware acceleration method*

The graphics chip that is now available in every computer can serve a further possibility for accelerating applications. These chips provides enormous computing power[2]. Originally, the graphics cards were used to accelerate applications that relied on graphical operations. DirectX or OpenGL serves an interface for the programmer to access the graphics card. Thus applications can be developed, which execute graphic operations more efficiently on a graphic chip than a conventional processor. In most cases, the graphical operations are used to display two or three dimensional objects. Other operations, such as matrix multiplication, must be performed on the CPU. DirectX and OpenGL are fixed to predefined functions and offer little flexibility apart from graphical operations. But with the development of Cuda by Nvidia it is possible to program graphics cards more versatile. In 2006 Nvidia released its "General Purpose Parallel Computing Architecure". With this programming model and tools provided by Nvidia, Nvidia GPUs can be used to create applications that use the GPU as a flexible accelerator [6].

But only the graphics chips from Nvidia support Cuda. Other graphics cards such as AMD cannot be programmed with Cuda. So the development of the OpenCL [7] standard started at the end of 2008. First developed by Apple, then by Khronos, the library is now available under version OpenCL 2.2. In cooperation with AMD, IBM, Nvidia and Intel, a library has been created which makes it possible to use different graphics cards from Nvidia. Under OpenCL, not only graphics chips can accelerate applications, CPUs or cell processors can also be used. To run OpenCL, the hardware manufacturer must implement the OpenCL standard on his device. This means that there may be Nvidia chips that support both Cuda and OpenCL, only one of the two, or no support at all. It is also not necessary that every AMD graphics card offers the possibility to program them with OpenCL.

## III. INTRODUCTION

The implementation and analysis presented in Section II serves as a reference, as the matrix vector multiplication takes up the most runtime. For comparison, a variant must be implemented on a CPU and a GPU.

The big advantage of the graphics cards over the FPGA is their comparable ease of use with OpenCL or Cuda. On the one hand the programmer does not have to learn a completely new language, since Cuda/OpenCL can be used to program in a subset of C++. The integration of the program parts into the overall application is also very simple. In summary, a simpler development process makes you more flexible than with an FPGA implementation.

### *A. Hardware accelerator*

Since the work involves embedded systems and in order to provide comparability to the FPGA, no normal graphics card as used in desktop/home computers can be used. These require considerably more power than conventional embedded systems. According to [8], a standard AMD RX 580[3] consumes about 200 watts under load alone. A technology must be used that can be embedded, but at the same time has a graphics chip that can provide enough computing power. Nvidia provides a good basis with its Jetsonboards. An ARM core with an integrated Nvidia graphics chip is installed on that boards. In March 2017 Nvidia brought an update to its Jetsonboards with the Jetson TX2. The Cortex-A57 got a slightly higher clock rate which is now 2 GHz, and it got 8 GB instead of only 4 GB of DDR4 RAM. But the big innovation is that the graphics chip consists of a total of 256 Cudacores of the Pascal architecture Nvidias. This also increased the clock rate of the graphics chip, but the overall power consumption of the system could be reduced again. According to Nvidia, the system typically consumes 7.5 watts under load. The Pascalchip of the board supports the CUDA architecture, which makes it easy to program. Thus the board is best suited for an implementation of the matrix vector calculation. It is designed for the embedded area and offers current hardware to compare it with the FPGA [9].

A limitation brings the Nvidia Board with it. The graphics chip must be programmed with CUDA. The alternative OpenCL is omitted because Nvidia does not support OpenCL for the board. Thus only a CUDA implementation and a pure CPU implementation can be realized in the context of this work.

### *B. General Purpose Computing on Graphics Processing Units*

GPGPU (**G**eneral **P**urpose Computing on **G**raphics **P**rocessing nits) is the general purpose calculation on graphics cards/processors. The graphics card can no longer only be used for graphical operations. With GPGPU it is possible to use graphics processors more versatile. Mathematically more complex problems can be solved more efficiently. Matrix multiplication is such a problem that can be solved by GPGPU [6].When programming the graphics chips, a concept is used which is known as Single Instruction, Multiple Data (SIMD). The advantage of a graphics chip over a normal CPU is that the same instructions, such as multiplications, are executed in parallel with different data. On the one hand, a graphics chip has several processor units on which instructions can be executed in parallel. On the other hand, each individual processor unit is able to execute several threads simultaneously [6].

In comparison to normal desktop CPUs, Graphics cards have a much larger number of processing units, which reaches into the three-digit range[4] in current models. To use and control this set of processors effectively, graphic chips are divided into structure elements. A graphics card usually has several Graphical Processing Clusters (GPC). Similar to multi-core CPUs, each individual GPC has the full range of functions of a graphics card. A GPC consists of several stream multiprocessors (SM), which in turn contain the actual thread

[2] Current desktop graphics cards, such as the Nvidia RTX 2080 TI, offer a theoretical computing performance of about 14.2 TFlops [5]

[3] Upper mid-range graphics card

[4] A Nvidia RTX 2080 TI, for example, has 4352 Cudacores. [10]

processors (TP). The thread processors are called Cuda cores at Nvidia and are the unit responsible for the execution of instructions [11].

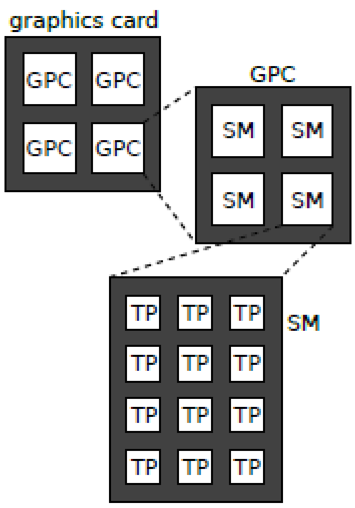

Fig. 2. Hierarchy Levels of a Graphic Chip [11]

A streaming multiprocessor, shown in Figure 3 of one of the GP104 GPUs from the Pascal architecture of Nvidia, basically consists of 128 Cudacores (TPs).

Such a **thread processor** has an arithmetic logic unit (ALU) and a floating-point unit (FPU), which is divided into different pipeline levels. The execution of instructions in the smallest unit af an GPU, however, is subject to some restrictions that do not exist in a multi-core CPU with its individual processor cores. Only the same instruction can be executed on several TPs in parallel. However, this instruction is executed by each TP on different data, whereby the SIMD principle is executed. The **warp scheduler** is the unit that reads instructions, decodes them and then distributes them to the cores. With an operating system, this distribution is usually controlled in software, here it is realized in hardware. Such a distribution is called warp and always contains 32 threads. Thus a warp scheduler instructs 32 TPs with one instruction on different data. In reality, however, only 16 TPs, i.e. a Half-Warp, are instructed to execute the same instruction. Later the other half of the warp is executed. However, in order for each of the 128 cores of the streaming multiprocessor to calculate simultaneously, a warp scheduler executes two half warps of different warps. In addition, there are not one but four warp schedulers per SM, each of which can instruct 32 TPs. The **LD/ST** units are used to execute one storage instruction per clock cycle. Thus a result can either be stored or a parameter can be loaded from an LD/ST unit. However, this memory addressing refers primarily to the L1 cache. The **SFU - Special Function Unit** is there to execute more complex instructions which the TPs do not support. These are instructions like sin(x) or exp(x). The **register** is available to the TPs as a fast memory and is divided among them. The **shared memory** is used to exchange data between the individual TPs [11].

## IV. IMPLEMENTATION

The evaluation program basically executes all the different variants one after the other in order to measure the runtime of each implementation for each execution. The variant for the CPU is executed first. But first the data of the matrix and the vectors have to be generated. The random data is stored in the vectors and the matrix. In the standard configuration, the matrix size is N x N N = 3. The individual vector and matrix

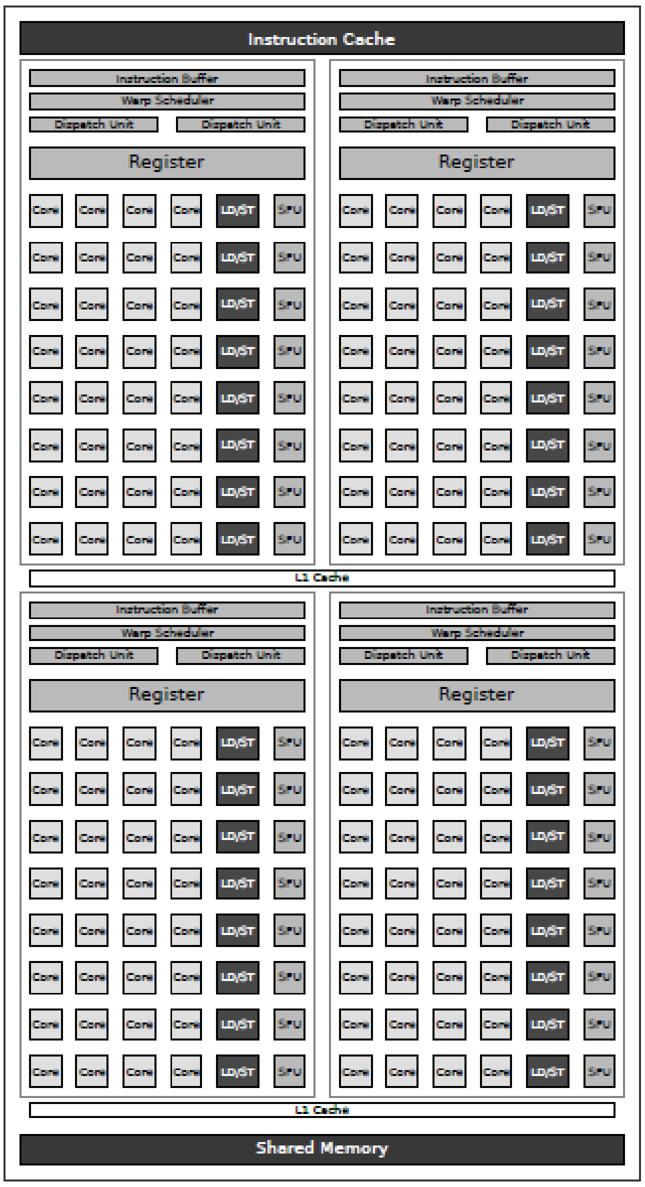

Fig. 3. Streaming multiprocessor [12]

entries are stored as complex numbers. The complex numbers are represented in the program by two double variables, one for the real part and one for the imaginary part. The various parameters (see Table 1) have been modified for the complete measurement series.

TABLE I
PARAMETERS FOR THE MEASUREMENTS

| Parameter | Range |
|---|---|
| Number format | complex numbers<br>Double (64 Bit)<br>Float (32 Bit)<br>Integer (32 Bit) |
| Matrix size $N \times N$ | $N = [1..32]$ |
| Number of calculations $c$ | $c = [1..8192]$ |

In order to be able to access the generated data from the GPU, it must be copied from the CPU into the memory of the GPU. In order to avoid time-consuming copying of the data, CUDA offers the possibility to request special memory from the CPU. This memory has the property that the host (i.e. the CPU) and the device (the GPU chip) have access. It is not necessary to explicitly send the data to the GPU. It is enough to copy the pointers to them into the memory of the GPU. This saves a lot of communication effort with large amounts of data and the programmer can leave the management of the memory to CUDA. After the CPU variant the GPU implementations are called. The host, the CPU, starts the kernel with the specified parameters for the number of threads per block and the number of blocks on the GPU. While the GPU executes the kernel, the CPU is able to perform calculations in parallel. However, if results from the GPU are required, a synchronization between processor and graphics card must be performed. In this paper, this synchronization is used to measure the execution times of the GPU. The CPU starts a kernel and does not process any commands until the graphics card has finished its calculation. Thus a time stamp is set directly before the kernel launch in order to form the difference with a second time stamp, which is set directly after the synchronization. With the CPU variant, the time stamps are set directly before and directly after the function is called. No communication is required here, as the CPU already has access to the data. In order to obtain representative measured values, each calculation is carried out 250 times in order to calculate the mean value. The processor was clocked to a constant 2GHz by a setting in the operating system in order to prevent large deviations in the measured values. The clock frequency of the graphics chip is also fixed. It clocks constantly with 1300MHz.

## *A. Variant 0 - CPU*

Figure 4 shows the implementation for the CPU. This variant always uses a thread of the CPU for execution, even if the number of calculations c or the matrix size N changes. Variant 0 is an implementation of the standard method to multiply matrices for the CPU. For the algorithm the matrix size n = N is given. The calculation is divided into three sections (S1-S3). In S1, the temporary memory for the result of S2 is initialized and set to zero. There must be memory for n results. In S2 the first part of the matrix vector calculation is executed. The vector v_1 is calculated with the matrix m in two interlaced loops. The n scale products from v_1 and the column vectors m_i of the matrix are stored temporarily in tmp. S3 is the section of the calculation where the scalar product tmp from S2 is scalar multiplied with the vector v_2. The result of the complete calculation is stored in r.

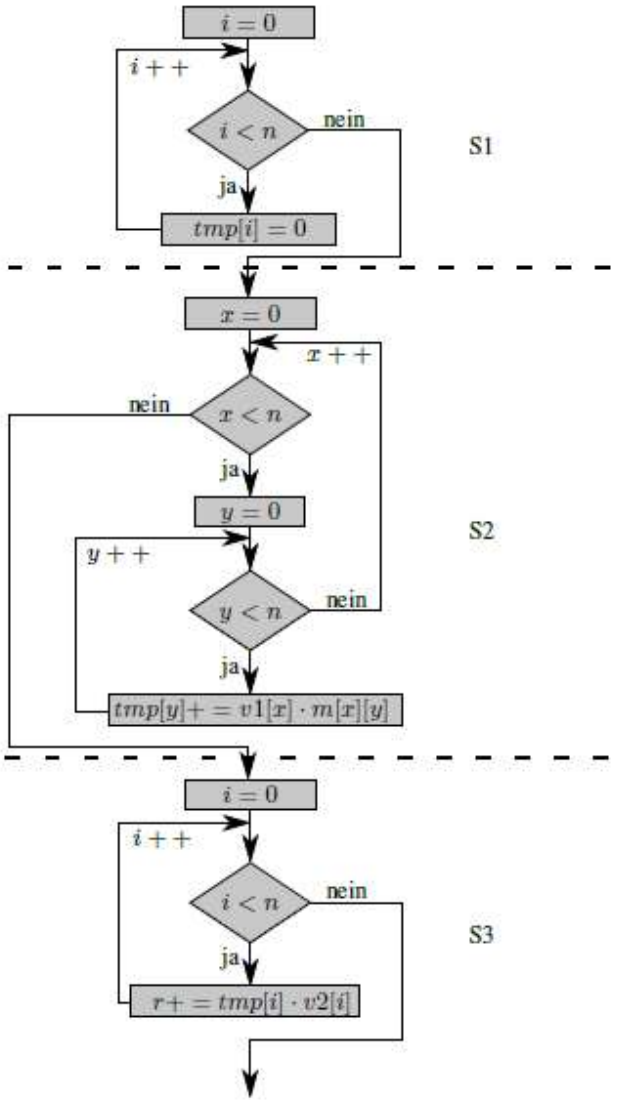

Fig. 4. Variant 0: One calculation

Figure 4 shows only one calculation c = 1. If c is increased by a multiple, variant 0 on the CPU is executed c times in succession. S2 has the most influence on the calculation, since the two nested loops have to be passed through n times. Thus, the implemented algorithm is in the complexity class of $O(n^2)$. If the number of calculations c is also considered, this has a linear influence on the runtime. When put together, the result is $O(n^2 c)$. For all variants it is important to be noted that as soon as a calculation operation with complex numbers is executed, a library was used. Nvidia provides some libraries when using Cuda, so the cuComplex library could be used. This library offers the possibility to use a data structure which maps complex numbers to two double variables. Furthermore, functions for adding, subtracting, dividing, multiplying and forming the absolute value on this data structure are offered. In the individual variants, only adding and multiplying are used.

### *B. Variant 1 - GPU*

A variant without synchronization on the GPU is the same as on the CPU. A thread on the GPU processes a calculation as shown in Figure 4. The difference to the CPU variant only occurs when more than one calculation c > 1 is to be executed. One thread per calculation is created on the GPU, which processes the calculation without synchronization and independently of each other. The runtime complexity for one calculation does not change compared to the CPU, it is still $O(n^2)$. However, the runtime for c calculations changes. The number of threads scales linearly to the number of calculations c. If the hardware resources allow it and the scheduler can split the threads, the calculations can be executed in parallel on the TPs of the graphics chip.

### *C. Variant 2 - GPU*

The difference to variant 1 occurs in S2. For A calculation with the matrix size N = 3 on the GPU is no longer only one thread but three responsible. First, the shared memory tmp must be initialized. This requires a memory of N • sizeof (used number format). The three threads all have the same instructions. In a loop, the scalar product is first calculated with the vector $v_1$ and the column vector $m_1 – m_3$ belonging to the thread. Then the respective part of the second scalar product tmp - $v_2$ can be calculated and stored by the threads in the respective shared memory. Then a synchronization of the threads must take place, since from this point the data parallelism is no longer given. For the sum formation, one of the threads goes through a further loop to obtain the result r. For several calculations the same applies as for variant 1. c times threads are created. However, unlike in variant 1, n threads are responsible for one calculation. This means that c • N threads are created if the matrix size N is variable and the number of calculations c is variable. The number of required threads thus scales linearly to both sizes. In so far as the scheduler has the resources available on the graphics card, part S2 on Figure 4 is no longer quadratic dependent. Therefore as long as the threads are executed in parallel on the TPs, there is a linear dependency.

### *D. Variant 3 - GPU*

In order to calculate as much as possible in parallel, a further dimension has been added to the shared memory in variant 3. Its size with $N^2$ • sizeof (used number format) is now no longer linear but quadratic dependent on the matrix size N. For the first part of the calculation (see Figure 4 S2), $N^2$ threads are required in order to calculate the multiplications of the first part. These $N^2$ threads have to be synchronized a first time after storing their result in shared memory. Theoretically, it is sufficient for this part to synchronize the N threads involved in the same scalar product. But since the processing sequences merge later and there is no possibility to synchronize individual threads of a block in Cuda, all $N^2$ threads of the block involved in the calculation are synchronized. After the first synchronization, N threads calculate the sum and then form the product of this and the entries of the second vector. The results of these calculations can then be stored in the shared memory again. The threads must then be synchronized again in order to calculate the final result using one thread to sum up, as in variant 1. By parallelizing each individual multiplication, the first part of the calculation should be independent of the matrix size in its runtime. However, since only one thread can calculate the sum for the scalar product at a time, the runtime of the complete variant 3 should be linearly dependent on the matrix size N. However, in these linear sections, the processing is only adding and no multiplication, which could indicate a lower runtime compared to variant 2. With several calculations c the implementation is similar to variant 2, only that here per calculation N times threads more are generated. Thus a linear dependency of the runtime on the number of calculations c should also arise with variant 3, but only under the condition that the graphics card has correspondingly resources available.

### *E. Variant 4 - GPU*

This variant is the same as variant 3, except that the two synchronizations are omitted to measure the effect they have on the runtime. Also in variant 4 the number of threads depends on the matrix size and the number of calculations, as it is the case in variant 3.

## V. Implementation

Figure 5 shows the difference of the runtimes between the variants of the GPU as a function of the number of calculations with complex numbers. All variants have a staircase. This means that for each variant, the runtime suddenly increases sharply after a certain number of calculations. Jumps at a certain multiple of c can be seen in the graphs. However, the jump is different in strength and after a different number of calculations, depending on which variant is considered.

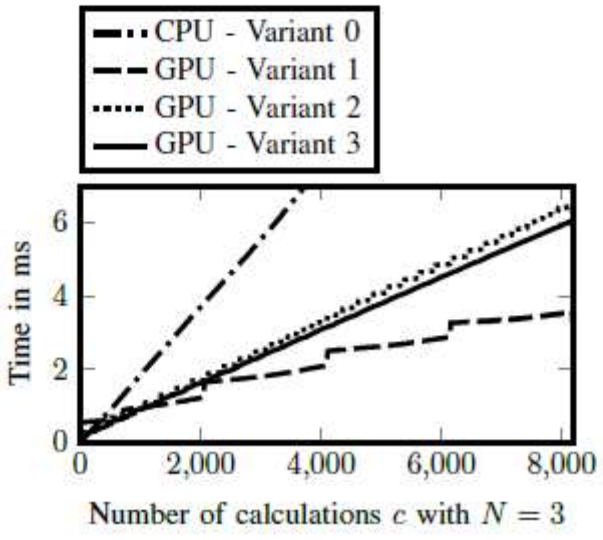

Fig. 5. The several GPU variants in comparison

In **variant 1**, which performs each individual calculation sequentially, but the different calculations in parallel, a jump of all 2048 calculations by about 0.4 ms can be seen. In between, the runtime increases only slightly. The difference between 2064 and 4096 calculations is about 0.45 ms. This staircase pattern can be traced back to the fact that the GPU can execute a certain number of calculations in parallel on the TPs of the graphics chip up to a certain number of calculations, which is equal to the number of threads in variant 1. Once all of them are busy, a TP must perform two calculations, which results in the jump. Between two jumps, however, the runtime does not remain constant, because due to the larger number of calculations, more threads have to be created and the number of memory accesses increases, which cannot be completely compensated by the warp scheduler. An irregularity can be seen in the graph at c = 688, because there is also a jump of about 0.18ms. Since this is not a multiple of 256, which would indicate a utilization of the GPU, which has 256 TPs, it can be that another resource is utilized on the GPU. Possibly the LD/ST units of the GPU are the limiting factor, which is not very important for more of the incoming calculations, because the warp scheduler can execute another instruction as soon as a memory access has to wait for data. The interval c = [2048..4096] shows a much larger increase after about c = 2048+688 = 2736, which is due to the same problem as the jump at c = 688, only that this delay can be better compensated by more calculations. In **variant 2**, which uses one synchronization and in this case three threads per calculation, jumps and thus the staircase pattern can also be recognized. However, these jumps are closer to each other, because all 256 calculations increase the runtime by about 0.18 ms. The distances are therefore narrower because more threads are needed per calculation. In contrast to variant 1, there is no noticeable increase between the jumps. In the interval c = [256..512] only a difference of about 0.02 ms was measured. Thus variant 2 does not have the problem that the TPs are not utilized and another resource is the limiting factor. The graph of **variant 3** shows an atypical trend. The staircase pattern is hardly recognizable. Only a jump at c = 448 of about 0.1 ms has occurred. This is repeated with twice the number of calculations c = 896 with approx. 0.08ms. Otherwise the course of the graph is approximately linear, since the jumps occur with increasing number of calculations in a minimum size or not at all. Figure 6 below illustrates the difference between the number formats used. The runtime of variant 2 can be seen as a function of the matrix size for one calculation. As expected, the runtime with complex numbers is significantly higher than with the simple data types integer, float and double. With a matrix size of N = 16, the calculation with complex numbers requires 5.9 times as much time as with integers. This is due to the more complex operations with complex numbers, where an addition does not consist of an addition of two numbers, but of the addition of imaginary parts and real parts. Likewise a complex multiplication is more complex, which consists of several multiplications and additions. However, no difference could be found between the use of integer and float numbers. The runtimes between these variants are almost identical. This fact can be traced back to the structure of the TPs. They have an arithmetic unit for integer and float numbers and can perform operations for both formats at the same speed. A calculation with double values needs on average 1.3 times as much time as with integers. Since a TP is only equipped with a 32 Bit floating point unit, the compiler either converts the 64 Bit Double values or uses two of the TPs to perform an operation with 64 Bit Double values.

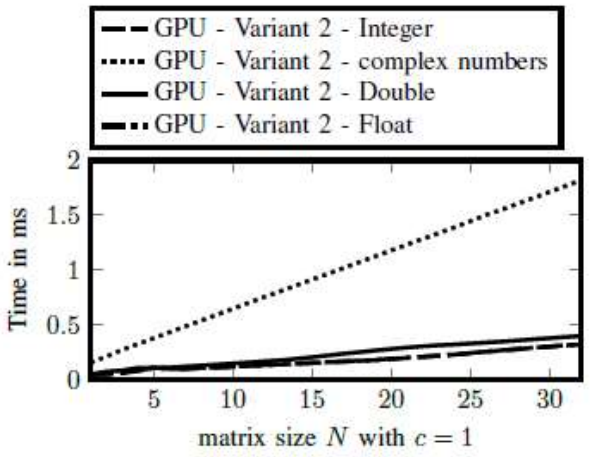

Fig. 6. Comparison of the different number formats with the variant 2 on the GPU

Figure 5 clearly shows that as the number calculations increases, the CPU takes considerably more time than the GPU variants.

TABLE II
RUNTIME AND SPEEDUP IN COMPARISON TO THE CPU WITH $c = 8192$ CALCULATIONS

| | Runtime | Speedup to variant 0 |
|---|---|---|
| **CPU - Variant 0** | 16.2128 $ms$ | 1 |
| **GPU - Variant 1** | 3.1645 $ms$ | $\sim 5.1$ |
| **GPU - Variant 2** | 6.1852 $ms$ | $\sim 2.6$ |
| **GPU - Variant 3** | 5.8117 $ms$ | $\sim 2.8$ |

With the fastest variant of the GPU, a speedup of ~5.1 could be achieved with c = 8132 calculations. The GPU is not only faster in the absolute runtime, with ~4.1 Mrd required clock cycles, the GPU also needs considerably less than the CPU with ~32.4 Mrd clock cycles. This results in a speedup of ~7.9. By executing instructions in parallel, the GPU can execute more instructions per clock cycle than the CPU, but only if the instructions contain the same operations but are to be executed on different data. However, enough data must be available for the GPU to take advantage of the parallelism of its architecture (see Figure 5). The CPU has up to c < 144 a faster runtime than variant 3 and is up to c < 160 faster than variant 2 of the GPU. Only from c > 336 variant 1 of the GPU is faster than variant 1 of the CPU. So for small amounts of data the CPU should be used instead of the GPU for this calculation. However, if the amount of data is large enough for the GPU to calculate efficiently in parallel, the GPU in all variants is faster than the CPU (c > 336). Since the change of the matrix size in the FPGA implementation has only a negligible small influence on the runtime, the GPU is clearly worse than the FPGA in this aspect. However, the GPU can reduce from a quadratic dependency to a linear dependency at runtime by parallelization. The FPGA has a runtime of ~2.1 ms at c = 92160 calculations, whereas the GPU at c = 8132 needs ~3.1ms. With an regression via the measured data, for variant 1, the runtime can be expressed by the following formula:

$$Variante\ 1:\ t(c) = 0.0004 \cdot c + 0.5713$$
$$t(92160) = 0.0004 \cdot 92160 + 0.5713 \approx 37.4\ ms$$

This results in a speedup for the FPGA design compared to the fastest variant on the GPU of ~ 17.8. The FPGA achieves this speed factor through pipelining, which means that the data throughput is extremely high despite the low clock rate. The GPU, which calculates with 13 times the clock frequency, needs considerably more time. Only a certain number of TPs are available for the calculation and the parallel calculation of these cannot make up the time as the pipeline effect that occurs in the FPGA. As mentioned above with regression via the measured data and with eliminating the frequency for each architecture, the CPU, the GPU and the FPGA, a formula, which represents the runtime t dependent on the number of calculations c and a used frequency f can be formed.

$$t_{FPGA}(c,f) = \frac{0.00002 \cdot c + 0.2616}{f} \cdot 100$$
$$t_{GPU\ Variante1}(c,f) = \frac{0.0004 \cdot c + 0.5713}{f} \cdot 1300$$
$$t_{CPU}(c,f) = \frac{0.002 \cdot c - 0.288}{f} \cdot 2000$$

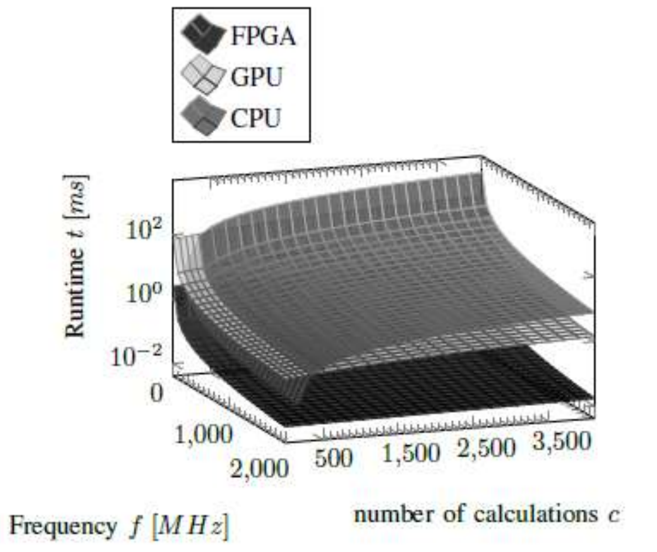

Fig. 7. Runtime $t$ dependent on the number of calculations $c$ and a used frequency $f$

With a speedup of ~17.8 the FPGA is faster than the GPU. As expected, a runtime with an FPGA cannot be achieved by a software solution. However, it should be noted that the development of a hardware solution requires more time and knowledge about the technology used. An already ~5.1 times faster solution than a processor, however, provides an implementation with a GPU which scales linear via the problem size by executing instructions on the TPs in parallel. A GPU can thus be used as a hardware accelerator, but only under a certain class of problems. These should be parallelizable by using the SIMD principle. Only then the threads can created for the GPU to execute in parallel. The parallelization of the problem should therefore be as data-independent as possible. A hardware acceleration by an FPGA is more independent of the type of problem. Using hardware description languages, an FPGA design can be perfectly configured to a problem. Thus, an FPGA is not only faster in absolute runtime, an FPGA implementation is also more efficient because it requires fewer clock cycles. From this it can be concluded that when it comes to meeting extremely tight deadlines, as it is often the case in real-time systems, the FPGA should be used as a hardware accelerator. If, however, the goal is to accelerate a problem with as little development effort as possible and to reduce the load on the main processor, a GPU solution can significantly reduce the runtime of a problem caused by data parallelism. If, however, only a few data (here calculations) are to be processed, a solution by the CPU is usually the fastest and in view of the development time the best solution.

## VI. Future Work

Since only the Nvidia Jetson TX2 board with 256 TPs was used for evaluation, the GPU variants could be tested on different graphics processors. Thus it can be determined what effects the number of SMs per GPU and TPs per SM has on the implementations. A possible option would be the Jetson AGX Xavier Board [13] which was released in September 2018. The number of graphic cores as well as the number of CPU cores and the main memory were doubled compared to the Jetson TX2. Published by Nvidia, a Deep Learning algorithm executed on the module had a maximum power consumption of 46 watts [14]. Furthermore, it can be investigated what influence the kernel size setting has on the runtime. This means that runtimes with different block and grid sizes could be measured and investigated.

## References

[1] Basiswechsel und Koordinatentransformation, https://elearning.\\physik.uni-frankfurt.de/data/FB13-PhysikOnline/lm_data/lm_8699/daten/ana2_la4.html.

[2] Matrizenoptik – Physik-Schule, https://physik.cosmos-indirekt.de/Physik-Schule/Matrizenoptik

[3] FPGA – Mikrocontroller.net,\\ https://www.mikrocontroller.net/articles/FPGA

[4] R. Schmidt, S. Blokzyl, W. Hardt: Hardware Acceleration for Beamforming Algorithms based on Optimized Hardware-/Software Partitioning (2018), https://www.ama-science.org/doi/10.5162/ettc2018/4.1

[5] Nvidia GeForce RTX 2070, 2080, 2080 Ti: Raytracing-Beschleuniger zu stolzen Preisen, https://www.heise.de/newsticker/meldung/Nvidia-GeFo rce-RTX-2070-2080-2080-Ti-Raytracing-Beschleuniger-zu-stolzen-Preisen-4142119.html

[6] CUDA – Thomas-Krenn-Wiki, https://www.thomas-krenn.com/de/wiki/CUDA

[7] OpenCL - Open Computing Language, https://www.uni-regensburg.de/EDV/kurs_info/brf09510/hpc/opencl/opencl.html

[8] Leerlauf runter, Last hoch - Radeon RX 580 und RX 570 im Test: AMDs Grafikkarten sind schneller und sparsamer - Golem.de, https://www.golem.de/news/radeon-rx-580-und-rx-570-im-test-amds-grafikkarten-sind-schneller-und-sparsamer-1704-127204-5.html

[9] NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge (Mar2017), https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[10] Nvidia GeForce RTX 2070, 2080, 2080 Ti: Raytracing-Beschleuniger zu stolzen Preisen, https://www.heise.de/newsticker/meldung/Nvidia-GeForce-RTX-2070-2080-2080-Ti-Raytracing-Beschleuniger-zu-stolzen-Preisen-4142119.html

[11] Gipp, M.: Online- und Offline-Prozessierung von biologischen Zellbildern auf FPGAs und GPUs (2012), http://archiv.ub.uni-heidelberg.de/volltex tserver/13349/

[12] GeForce GTX 1080 Whitepaper p. 52

[13] NVIDIA jetson AGX xavier developer kit now available, https://news.developer.nvidia.com/nvidia-jetson-agx_xavier-developer-kit-now-available/

[14] Jetson AGX xavier: Deep learning inference benchmarks, https://developer.nvidia.com/embedded/jetson-agx-xavier-dl-inference-benchmarks