Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science

GPGPU acceleration

Internship report

Lucas Bekker

Supervisors:

prof.dr.ir. E. Harald van Brummelen

v1.0

Eindhoven, August 2018

Contents

List of Acronyms

GPGPU	General-purpose computing on graphics processing units
TU/e	Technische Universiteit Eindhoven
MEFD	Multiscale Engineering Fluid Dynamics
GPU	Graphics processing unit
CPU	Central processing unit
SRAM	Static random access memory
DRAM	Dynamic random access memory
SoC	System on a chip
HDD	Hard disk drive
SSD	Solid state drive
FEM	Finite element method
FP	Floating point
FLOP	Floating point operations
FLOPS	Floating point operations per second
FMA	Fused multiply add
FMA3	Fused multiply add version three
AVX	Advanced vector extensions
AVX-512	Advanced vector extensions 512
AVX2	Advanced vector extensions two
SMT	Simultaneous multithreading
ISA	Instruction set architecture
uarch	Microarchitecture
RAM	Random access memory
ALU	Arithmetic logic unit
AGU	Address generation unit
FPU	Floating point unit

SIMD	Single instruction multiple data
OS	Operating system
HPC	High performance computing
MKL	Intel math kernel library
DIMM	Dual in-line memory module
ECC	Error-correcting code
PCI	Peripheral component interconnect
PCI-X	Peripheral component interconnect extended
PCIe	Peripheral component interconnect express
AGP	Accelerated graphics port
NIC	Network interface controller
SAS	Serial Attached SCSI
SCSI	Small computer system interface
NVMe	Non-volatile memory express
DMI	Direct media interface
USB	Universal serial bus
SATA	Serial AT attachment
QPI	Intel quickpath interconnect
UPI	Intel ultrapath interconnect
CUDA	Compute unified device architecture

Chapter 1

Introduction

The goal of this internship is to investigate the possibilities of GPGPU acceleration of numerical simulations and subroutines in use at the MEFD group. GPGPU acceleration leverages the computational power of the GPU to perform mathematical calculations that would otherwise be performed by the CPU.

GPGPU is a hot topic at the moment, artificial intelligence and crypto currencies have taken a huge flight since they adapted to leverage the power of GPU's. As figure 1.1 shows, the peak double precision FLOPS and memory bandwidth of GPU's has grown to be much larger than CPU's can provide.

A reduction in runtime of numerical simulations has numerous advantages, the most obvious one is the ability to run larger simulations in the same runtime. Training of deep learning neural networks is a prime example. Much of the theory behind these applications was already available in the eighties, but the compute capabilities at the time where insufficient for practical use.

The productivity of researchers also improves when they have access to increased computational power. It allows them to run more and better simulation, proving an enhanced insight into the results.

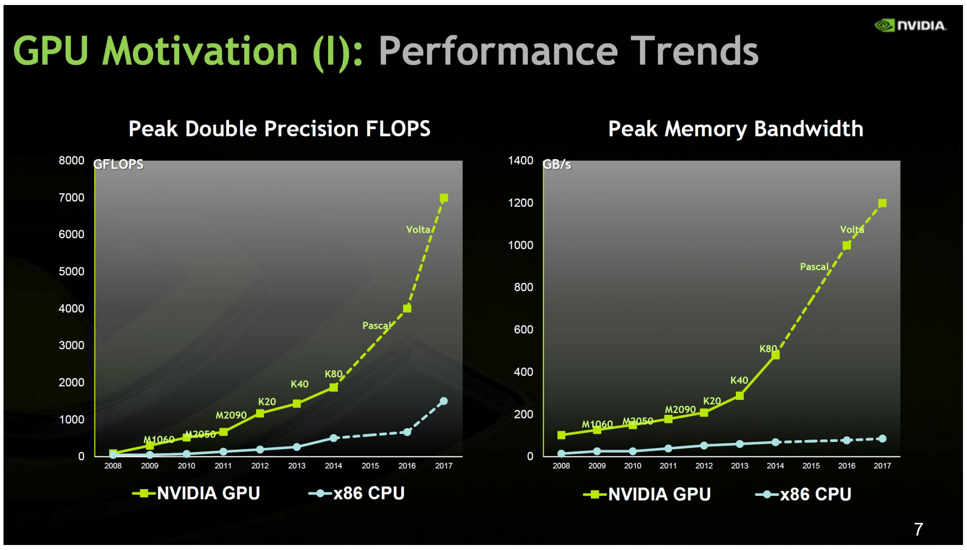

Figure 1.1: Floating point and bandwidth performance comparison provided by NVIDIA.

1.1 The MEFD group

The website of the Department of Mechanical Engineering faculty of the TU/e gives the following description of the MEFD group:

The MEFD section focuses on the development, analysis and application of mathematical-physical models and advanced numerical techniques for multiscale flow problems in engineering applications, with particular emphasis on flow problems in the transitional molecular/continuum regime and auxiliary field interactions. The research in the section has an underpinning and methodological character, while maintaining a strong connection to applications in the high-tech industry and in other sections.[1]

The work within the MEFD group can be roughly divided into 5 categories:

- Research into various models of physics and engineering based problems.
- Investigating the mathematical aspects of the aforementioned models.
- Evaluating and enhancing discretization techniques and their relation to the models for numerical evaluation.
- Implementing the advancements in discretization methods into software (Nutils).
- Looking into various algorithms for solving the resulting linear system of equations.

The research within the MEFD group into iterative methods, stems from the problems encountered when using the more traditional direct solving techniques in their applications. Methods like LU decomposition can cause excessive run-times or residual errors, making them unsuitable in certain instances. A similar problem can arise when the simulations grow in size. The run-times can grow out of control when the amount of FLOP's required to evaluate the simulations scales badly with the dimensions of the simulation.

The most obvious solution to this problem is to buy computers with increased computational power, but the recent movement towards GPGPU accelerated computers also requires a paradigm shift in programming techniques to leverage their capabilities to the fullest. The MEFD group has recently gained access to a GPGPU enabled server via the Rare Trans project of ASML in conjunction with the Energy Technology group, of which the MEFD group is a part. Its adoption for the simulations of the MEFD group is (at the time of writing) at an early stage.

[1] *Department of Mechanical Engineering, Multiscale Engineering Fluid Dynamics.* TU/e. 2018. URL: `https://www.tue.nl/en/university/departments/mechanical-engineering/research/research-groups/multiscale-engineering-fluid-dynamics/`.

1.2 The internship project

A lot of the numerical methods and simulations that are developed by the MEFD group are based on FEM (or related methods) and often require solving ill-conditioned block-sparse linear systems. These types of applications require relatively large amounts of RAM and double precision floating point data, resulting in additional difficulties when making the transition to GPGPU.

This internship will lay the foundations for a discussion of these difficulties by providing the required background on floating point arithmetic, Amdahl's law and the workings of computers. A high level overview of computers and their history pertaining to scientific computing will be provided, followed by a description of the somewhat more low-level workings of a personal computer.

The final chapter of this report will provide a brief summary of the topics of the master thesis project and their associated challenges.

Chapter 2

Floating point data

Floating point data is a common way to store real numbers . Many different types of floating point data types exist, but the most relevant types for scientific computing are:

- Double (FP64)
- Single (FP32)
- Half (FP16)

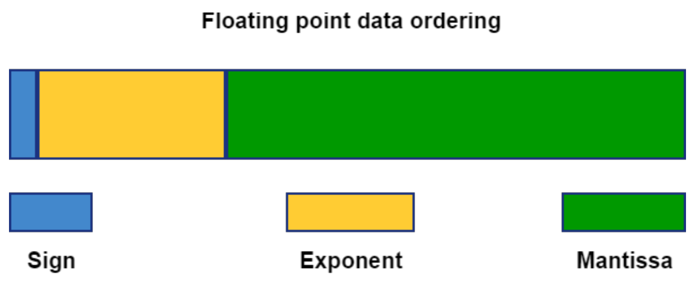

Figure 2.1: Floating point data structure.

Floating point data contains three parts, the sign, the exponent and the mantissa. The sign and exponent are both stored as 2-base little-endian (most significant bit to the left) numbers. The mantissa is treated slightly differently. It is stored as a 2-base big-endian number (most significant bit to the right) with an implicit least significant bit equated to 0.

2-base number conversion

Equation 2.1 provides an example conversion from the binary representation to the standard 10-base representation.

$$\begin{aligned} \text{2 base sign/exponent} &= 0111 \\ \text{10 base sign/exponent} &= (8 \times 0) + (4 \times 1) + (2 \times 1) + (1 \times 1) = 7 \\ \\ \text{2 base mantissa} &= 0111 \\ \text{10 base mantissa} &= (1 \times 0) + (2 \times 0) + (4 \times 1) + (8 \times 1) + (16 \times 1) = 28 \end{aligned} \tag{2.1}$$

2.1 Calculating floating point value

Equation 2.2 provides the generic conversion formula for FP data.

$$\text{Value} = -1^{Sign_2} \times 2^{Exponent_{10} - Bias_{10}} \times (1 + 1/(Mantissa_{10})) \tag{2.2}$$

Bias is a data type specific number to "center" the exponent range around 0.

Double (64 bit)

Common format for sensitive mathematical equations, like solving ill conditioned systems. It is also the default numerical data type of MATLAB.[1]

- Bias: '1023'
- Sign: 1 bit
- Exponent: 11 bits
- Mantissa: 52 bits
- Resolution: $(1/2)^{52} = 2.2204e - 16$
- Exponent range: $2^{([-1023,1024])}$

Single (32 bit)

Original floating point data type used in 32 bit computers, provides sufficient resolution for many mathematical operations.[2]

- Bias: '127'
- Sign: 1 bit
- Exponent: 8 bits
- Mantissa: 23 bits
- Resolution: $(1/2)^{23} = 1.1921e - 07$
- Exponent range: $2^{([-127,128])}$

[1] *Double-precision floating-point format.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`.

[2] *Single-precision floating-point format.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Single-precision_floating-point_format`.

Half (16 bit)

Relatively coarse approximation of a real number, but sometimes good enough. A great example of applications that usually don't require more precision are deep neural networks.[3]

- Bias: '15'
- Sign: 1 bit
- Exponent: 5 bits
- Mantissa: 10 bits
- Resolution: $(1/2)^{10} = 9.7656e - 04$
- Exponent range: $2^{([-15,16])}$

[3] *Half-precision floating-point format.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Half-precision_floating-point_format`.

Chapter 3

Floating point arithmetic

3.1 Fused Multiply Add

The prolific nature of the dot product in vector mathematics has lead to the development of hardware dedicated to the task, performing a multiplication and addition in a single step:

$$\mathrm{A} = \mathrm{A} + \mathrm{B} \times \mathrm{C} \tag{3.1}$$

It should be noted that the FMA operation can be used to emulate (may take more then a single instruction) the basic operations:

- Addition
- Subtraction
- Multiplication
- Division
- Raising to a power

As such, most modern computers rely on FMA hardware for all their floating point calculations. The low level hardware required to perform the FMA calculation, differs for every floating point format (double, single or half). This makes it a common practice to provide separate hardware blocks for each floating point format.

Chapter 4

Theoretical parallel speedup

Applying parallel programming techniques to accelerate an algorithm is a very common task. This section will discuss some of the available theory for making predictions about the maximum speedup that can be achieved.

4.1 Amdahl's law

Amdahl's law relates the maximum achievable speedup of a task to the amount of time that the task executes elements that can be parallelized.[1] This is based on two assumptions:

- The task can be divided in a serial part and a parallel part.
- The parallel part scales linearly with the amount of available parallel nodes.

$$\begin{aligned} \mathrm{T}_{total} &= \mathrm{T}_{serial} + \mathrm{T}_{parallel} \\ \text{Speedup}(k) &= \frac{\mathrm{T}_{total}}{\mathrm{T}_{serial} + (\frac{\mathrm{T}_{parallel}}{k})} \end{aligned} \tag{4.1}$$

$$k := \text{number of parallel nodes}$$

As an example, take a task that consist of 95% parallelizable work. The maximum speedup can never exceed 20, no matter how much parallel nodes are employed.

4.1.1 Parallel slowdown

Parallel slowdown is an effect that can plague certain parallel tasks.[2] Scaling the task beyond a certain amount of parallel nodes causes a runtime increase instead of the expected decrease. This is typically associated with a communications bottleneck, where the time required for communication between the parallelly executed parts grows faster than can be compensated by dividing the workload among the parallel nodes.

[1] *Amdahl's law*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Amdahl%27s_law`.
[2] *Parallel slowdown*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Parallel_slowdown`.

4.2 Compensated Amdahl's law

Amdahl's law assumes that dividing the parallel part of the task among many nodes doesn't inflict any sort of runtime penalty. That assumption is certainly reasonable for embarrassingly parallel workloads, but is not always correct outside of the embarrassingly parallel case.

Amdahl's law can be compensated for this runtime penalty by specifying and additional part of the task, called $\mathrm{T}_{overhead}(k)$, which is modelled as a function of the amount of nodes:

$$\begin{aligned} \mathrm{T}_{total} &= \mathrm{T}_{serial} + \mathrm{T}_{parallel} \\ \text{Speedup}(k) &= \frac{\mathrm{T}_{total}}{\mathrm{T}_{serial} + (\frac{\mathrm{T}_{parallel}}{k}) + \mathrm{T}_{overhead}(k)} \end{aligned} \tag{4.2}$$

$$k := \text{number of parallel nodes}$$

The exact relation between $\mathrm{T}_{overhead}(k)$ and the amount of nodes is application specific, but two general modelling assumptions seem reasonable:

- There is no overhead when only one node is available.
- $\mathrm{T}_{overhead}(k)$ consists of a parallelizable and non-parallelizable part.

The first assumption is trivial. The second one states that the tasks constituting the overhead can consist of elements that need to be executed sequentially and elements that can be executed concurrently.

4.2.1 Modelling parallel slowdown

Parallel slowdown can be modelled with the compensated Amdahl's law. When it is indeed caused by a communications bottleneck, it seems reasonable to assume the following things:

- The overhead consists exclusively of non-parallelizable tasks.
- The overhead scales linearly with the amount of parallel nodes.

Communication between multiple nodes is usually dominated by non-parallelizable tasks, because it often involves some form of causality. The choice of scaling is debatable, but less then linear scaling is very unlikely, whereas more than linear scaling is very common.

The predicted speedup would in this case be:

$$\begin{aligned} \mathrm{T}_{total} &= \mathrm{T}_{serial} + \mathrm{T}_{parallel} \\ \text{Speedup}(k) &= \frac{\mathrm{T}_{total}}{\mathrm{T}_{serial} + (\frac{\mathrm{T}_{parallel}}{k}) + ((k-1) \times \mathrm{T}_{overhead})} \end{aligned} \tag{4.3}$$

$$k := \text{number of parallel nodes}$$

4.2.2 Compensated example

An example will be discussed to illustrate the consequences that the overhead compensation of Amdahl's law can have. Like the previous example, the parallel part of the task is set at 95%. The relation for $T_{overhead}(k)$ is taken from the communications bottlenecked case, with a constant factor of 1% of T_{total}.

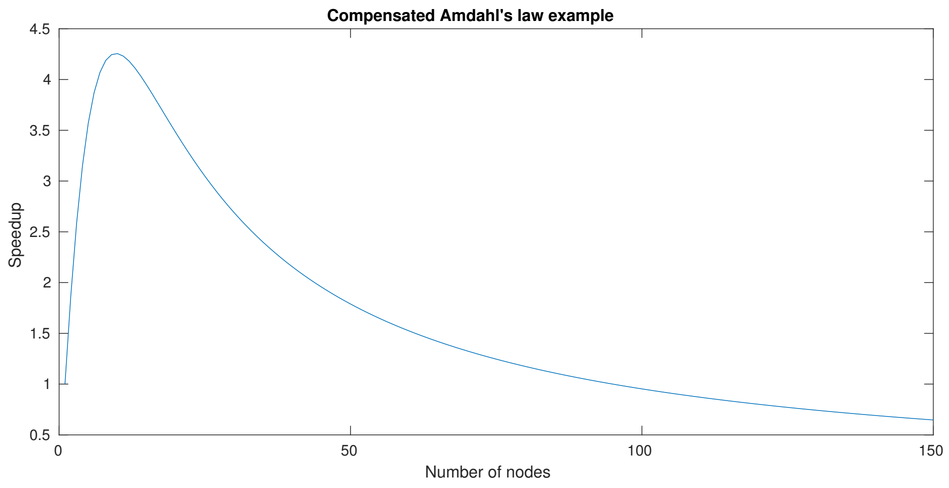

Figure 4.1: Task: 95% parallelizable, 5% serial and 1% overhead.

The maximum speedup of about 4.255 is achieved using 10 parallel nodes, a far cry from the maximum of 20 that the uncompensated Amdahl's law predicted. Using 150 nodes results in a speedup of about 0.65, which is actually a significant performance decrease.

4.3 Experimental verification

Accelerating iterative solvers by dividing the workload among parallel nodes is a common practice. However, there is a general rule of thumb that the amount of parallel nodes should not be to large in these cases, as the communication between the processes quickly becomes excessive.

Zuberek and Perera investigated the scaling behaviour of distributed iterative solvers, focusing on the Gauss-Seidel method.[3] They provide experimental data on the scaling behaviour for three kind of sparsity patterns:

$$
\begin{bmatrix}
* & * & & & & & & & & \\
* & * & * & & & & & & & \\
 & * & * & * & & & & & & \\
 & & * & * & * & & & & & \\
 & & & * & * & * & & & & \\
 & & & & * & * & * & & & \\
 & & & & & * & * & * & & \\
 & & & & & & * & * & * & \\
 & & & & & & & * & * & * \\
 & & & & & & & & * & *
\end{bmatrix}
\quad
\begin{bmatrix}
* & & * & & & & & & & \\
 & * & & * & & & & & & \\
* & & * & & * & & & & & \\
 & * & & * & & * & & & & \\
 & & * & & * & & * & & & \\
 & & & * & & * & & * & & \\
 & & & & * & & * & & * & \\
 & & & & & * & & * & & * \\
 & & & & & & * & & * & \\
 & & & & & & & * & & *
\end{bmatrix}
\quad
\begin{bmatrix}
* & * & & & * & & & & & \\
* & * & * & & & * & & & & \\
 & * & * & * & & & * & & & \\
 & & * & * & * & & & * & & \\
* & & & * & * & * & & & * & \\
 & * & & & * & * & * & & & * \\
 & & * & & & * & * & * & & \\
 & & & * & & & * & * & * & \\
 & & & & * & & & * & * & * \\
 & & & & & * & & & * & *
\end{bmatrix}
$$

(**1**) (**2**) (**3**)

Figure 4.2: Density: (1) = 0.015, (2) = 0.015, (3) = 0.025
(Taken from Zuberek and Perera)

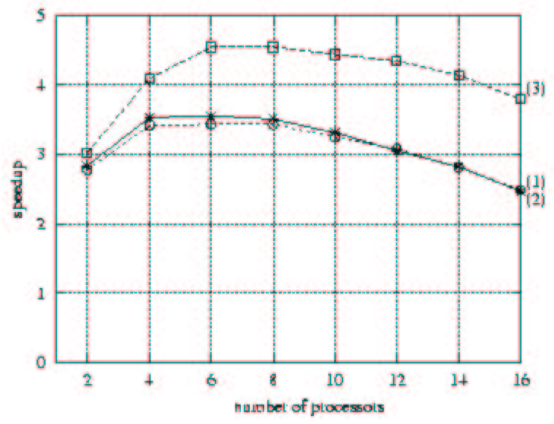

Figure 4.3: Speedups of distributed iterative solvers.
(Taken from Zuberek and Perera)

Figure 4.3 shows three things:

- Parallel slowdown occurs for all three matrices when using more then 8 processors.
- Sparsity pattern has little effect on the speedup.
- Density has a significant effect on the speedup.

Worth mentioning is the questionable speedup at 2 processors, having a value of about 3 for all three curves. This is odd because Amdahl's law states that the speedup can never exceed the amount of parallel nodes. It might be explained by some caching effect, where the complete data set doesn't fit in the cache of a single processor, but does fit in the cache of two processors. No mention of this was made by Zuberek and Perera.

[3]W.M. Zuberek and T.D.P. Perera. 'Performance Analysis of Distributed Iterative Linear Solvers'. In: *7th WSEAS Int. Conf. on Mathematical Methods and Computational Techniques in Electrical Engineering* (), pp. 194–199.

4.3.1 Fitted model results

To evaluate the performance of the compensated Amdahl's law, a comparison is made with the experimental data of Zuberek and Perera.

The model represented by equation 4.3 was selected for the comparison, because it is expected that the parallel slowdown was caused by a communications bottleneck. $T_{parallel}$ was set to 100% because the algorithm used by Zuberek and Perera contains no serial part. Some experimenting provided the values of $T_{overhead}$:

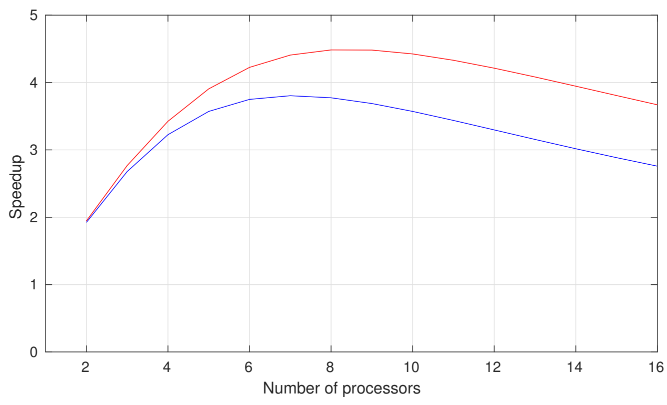

Figure 4.4: Red: $T_{parallel} = 100\%$, $T_{overhead} = 1.4\%$
Blue: $T_{parallel} = 100\%$, $T_{overhead} = 2.0\%$

The results are not too bad. The behaviour in the region between 2 and 6 processors is a little different, but speedups exceeding the number of parallel nodes cannot be explained with Amdahl's law. Some unknown effect must have had a significant influence on the results of the experimental data, but it is generally very uncommon to see this kind of behaviour.

The value of the maximum speedup and the amount of processors required to achieve said speedup are relatively accurate. The experimental results show a slightly less linear and steeper descend then the model, but this just means that the assumption that the overhead scales linearly with the amount of parallel nodes is not completely true. This reflects the discussion of the model provided at section "Modelling parallel slowdown".

The red line models the situation of matrix (3), with a density of 0.025. The blue line models the two other cases with the lower density of 0.015. The higher density situation has a lower overhead, 1.4% versus 2.0%. This appears reasonable, because the higher density results in a higher runtime per processor, while the communication overhead remains unaffected. As $T_{overhead}$ is modelled as a fraction of T_{total}, this lower $T_{overhead}$ is to be expected.

Chapter 5

Computer models

5.1 Personal computer model

The traditional personal computer was designed to perform many different operations on a relatively small amount of data. Because parallelising many different operations on the same data is very hard to impossible, sequential execution of the operations has long been (still is) the norm. The result is a combination of a few very fast execution units in the CPU and a "pyramid shaped" memory/storage system.

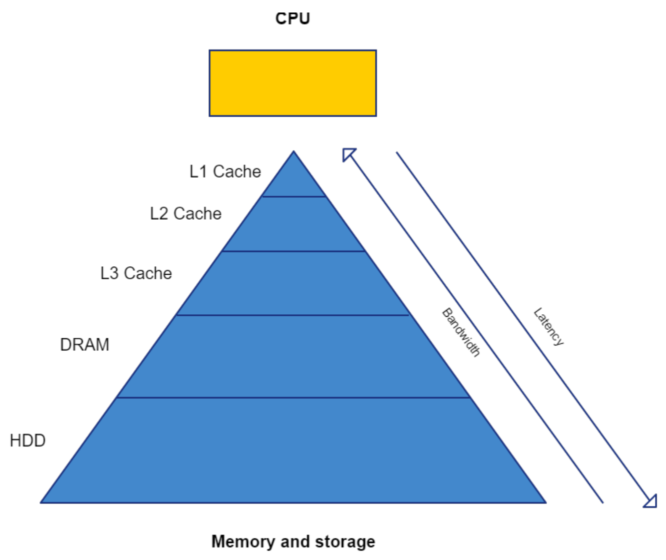

Figure 5.1: Abstract representation of the traditional personal computer.

Pyramid shape

Creating fast memory is expensive and not all the available data needs to be instantly accessible. Production cost savings led to a small amount of very fast memory close to the execution units, called "Cache". A larger amount of significantly slower memory, called "RAM", acts as the data overflow buffer of the cache. The lowest layer of memory consists of non-volatile storage, often in the form of an HDD or SSD, which contains the data that the needs to survive a power cycle.

As CPU's grew more capable, and the way in which they where used, changed over the years, additional layers (L2 and L3) of cache where added to accommodate.

5.2 Bandwidth

Bandwidth is one of two measures of the "speed" of memory, but bandwidth is a broader concept. Bandwidth is a measure of the maximum amount of data that can be transferred between a sender and a receiver within a time-frame.[1] Data transfer channels are usually bi-directional, making the distinction between sender and receiver less relevant.

High data transfer rates are desirable, but like storage capacity, are costly to achieve. The traditional computer model benefits most from high memory bandwidth at the cache level and becomes less important towards the bottom of the "pyramid".

The most common measure for bandwidth is bytes per second, a table of derived units and their meaning is provided by table 5.1. A bit is a single one or zero and a byte is a set of 8 bits.

Bandwidth

kilo	mega	giga	terra				bits per second
Kb/s	Mb/s	Gb/s	Tb/s	1.000	1.000.000	1.000.000.000	1.000.000.000.000
KB/s	MB/s	GB/s	TB/s	8.000	8.000.000	8.000.000.000	8.000.000.000.000
Kib/s	Mib/s	Gib/s	Tib/s	1.024	1.048.576	1.073.741.824	1.099.511.627.776
KiB/s	MiB/s	GiB/s	TiB/s	8.192	8.388.608	8.589.934.592	8.796.093.022.208

Table 5.1: Acronyms to the left and their numerical values to the right.

5.3 Latency

Memory latency is a highly multifaceted topic, which is largely beyond the scope of this text, but should be understood at a global level to appreciate the performance impact on certain algorithms. Latency, or access time, is defined as the amount of time between a request to a system and the response of the system.[2] Memory latency is the second important measure of the "speed" of memory.

In order to understand memory access time, one must first (roughly) understand how memory is accessed. Computer memory is a highly structured and complex system, with many different layers and subsystems (cache, RAM, HDD). The steps that are required to access data from computer memory are dictated by the structure of the global computer memory system, but also by the local structure of the memory subsystem that contains the data. These local structures can vary pretty significantly from subsystem to subsystem, but they all "group" their resources into "memory blocks", which contain the actual data. The nature of these "blocks" is defended by the precise structure of the specific memory subsystem.

When the user makes a request to access certain (parts) of data, a list of "storage addresses" is generated. This list of "storage addresses" maps to the memory subsystem and corresponding memory blocks that contain the requested data. This list of storage addresses is passed through to the relevant memory subsystem and then gets processed. This results in the data contained in the memory blocks being presented to one or more of the outputs of the memory subsystem in a serial fashion.

[1] *Bandwidth (computing).* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Bandwidth_(computing).
[2] *Latency (engineering).* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Latency_(engineering).

The time between the arrival of the storage addresses list and the moment the first block of data is presented, is called latency of that memory subsystem. The total latency of a data access request is the amount of time from the moment a ”gather” instruction is issued till the moment that the data is ”gathered” (available for manipulation).

Latency is usually measured in mili/nano seconds, but can also be provided in cycles. This is equivalent to time, because each cycle takes a set amount of time (dependant on operating frequency).

5.4 Scientific computing

The traditional personal computer model is optimized to perform many different operations on a small amount of data, but most workloads of a scientific computing nature are characterized by many similar operations on a large amount of data. Take for example a matrix-matrix multiplication, which consists of many floating point multiplications and additions, operating on two large amounts of data. This difference doesn't exclude scientific computing related tasks from running on personal computers, but the (personal computer) hardware is not tailored towards the task, resulting in performance that leaves a lot to be desired.

As the personal computer became faster, consumers where interested in using their computer for more than just office related workloads. Content creation like photo and video editing, but also gaming, are prime examples of these more modern tasks for the personal computer. These tasks are like scientific computing tasks, in the sense that they involve floating point operations on large data sets. This resulted in the development of much more powerful graphics cards, specifically for applications that required manipulations on graphical data sets, like decoding (viewing) video files or playing video games.

Meanwhile, scientific applications did not really see the development of specialized hardware. The only way to scale the performance of these tasks was to divide the workload among many computers organized in ”clusters”, relying on hardware that was developed mostly for hosting large relational databases. Database related workloads require fast non-volatile storage and the ability to handle many different requests. These requirements are similar to those of scientific computing, but are still very different in key aspects. Firstly, a request to a relational database does not require floating point operations. Second, the scale of relational databases often approach the terabyte range, making the capacity of RAM woefully insufficient. Typical medium to large scale scientific computing workloads don't exceed more than a couple of gigabyte of data, and as such are best stored in RAM.

The big game changing moment for scientific computing came when certain scientists realized that graphics cards, developed for consumers, could be very suitable for the needs of scientific computing. The only real problem was that the applications had to be programmed using graphics API's, like DirectX, making the process of writing simulations very difficult and restrictive. One of the main producers of graphics cards, NVIDIA, saw this problem and decided to alleviate some of these difficulties by making (relatively) slight modifications to graphics cards. The result, CUDA cores and CUDA toolkit, allowed for much easier and less restricted access to the resources provided by graphics cards for scientific computing workloads.

5.5 Personal vs Massively Parallel computer model

The Massively Parallel computer model differs from the traditional personal computer model in many ways because it focuses on simultaneous execution of computations.[3] To understand why parallel execution is fundamental for scientific computing, a very low level problem called the "power wall" should be explained.

CPU's and GPU's are made from transistors, which can be switched "on" or "off". A transistor is on when electrical current can flow through it, making it conductive. A transistor is off when electrical current cannot flow through it, making it non-conductive (hence the term semiconductors). Switching the state of a transistor requires a small amount of electrical current, the energy of that current needs to be dissipated in the form of heat. When a transistor is required to switch ever faster, because the performance of a CPU (or GPU) scales linearly with it's operating frequency, the current required to switch the state of the transistor also increases. The kicker is that the amount of current required to switch the state of a transistor scales quadratically with the switching frequency (caused by parasitic capacitance at the transistor). The final result is that attempts to increase the performance of a CPU or GPU by increasing its operating frequency are frustrated by a quadratic increase in power consumption and heat production, which at some point makes increasing the operating frequency impossible. This limit is called the "power wall".

Scaling the performance of a CPU or GPU by increasing the amount of execution units or cores has no such problems, but faces other difficulties, the biggest of which is causality. When the result of problem X is (deterministically) dependent on the result of problem Y, it follows that problem Y should be solved before problem X can be solved. These types of situations are extremely difficult (or impossible) to accelerate with parallel execution.

Many scientific computing workloads, like the aforementioned example of matrix-matrix multiplication, are known to be "embarrassingly parallel".[4] This means that parallel execution isn't hindered by causality related problems and as such yields the greatest performance (per Watt) on massively parallel systems, like GPU's.

[3] *Massively parallel.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Massively_parallel`.
[4] *Embarrassingly parallel.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Embarrassingly_parallel`.

Chapter 6

Central processing unit

6.1 Basics of a CPU

Processors, or CPU's, are the "beating heart" of a computer, and as such, perform many more tasks than the ones that will be discussed in this text. The focus lies on the floating point calculation capabilities and the memory subsystems, as well as the interfaces.

The traditional CPU[1] is an integrated circuit that executes the logic, arithmetic , input/output (I/O) and control operations that are prescribed by software running on the computer. As time passed, many other subsystems of computers got integrated into the processor package (die), making the functions that the traditional CPU performs only a subset of all the functions that a modern processor performs. A "core" of a modern processor is a separate unit that performs all the tasks of a traditional CPU.

Modern processor cores are very diverse, complex and multi-faceted, varying wildly with ISA, microarchitecture, intended platform and manufacturer. A discussion about CPU's that would include all these variations would be impossible, necessitating a confinement. This discussion will try to be as generic as possible, but the focus lies on an Intel based server CPU of the "Skylake-SP" microarchitecture.

Intel dominates the PC/laptop as well as the HPC/Supercomputer CPU market, making the restriction towards an Intel x86-64 based CPU justified. Considering the fact that almost all Laptops and workstations contain CPU's that are (to a varying extend) derived from their server oriented counterparts, focusing the discussion around a server CPU seems logical as well. The Skylake-SP microarchitecture was chosen because it is very recent (at the time of writing), contains some very significant advancements for scientific computing workloads and is used in a cluster available to the MEFD group at the TU/e.

[1] *Central processing unit.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Central_processing_unit.

6.2 System on a Chip

Modern processors are best described by the "System on a Chip"[2] moniker, containing many of the core components of a computer. As such, most contain the following subsystems:

- Cores
- Memory controller
- Cache
- Interfaces
- Graphics (included in most consumer oriented CPU's)

6.3 Instruction set architecture

An Instruction Set Architecture is an abstract model of a computer and contains a collection of machine instruction definitions.[3] Examples of common ISA's are x86-64, x86 and ARM, with x86-64 being the most common ISA for CPU's in servers, as well as consumer oriented computers.

An ISA is one of the most important aspects of a CPU, because it forms the link between software and hardware. ISA's where introduced to make programming software easier, which could now be written in terms of ISA instructions in stead of low level machine code.[4] This made it possible to execute the same computer program on different computers, without any modification of the code.

An implementation of an ISA, called a microarchitecture (uarch),[5] is the hardware based realization of these machine instruction definitions (disregarding microcode[6]). Any specific uarch can also support extensions to its ISA, common examples are VT-d, AES-NI, SSE4 and AVX2. These extensions are additions to the abstract computer model of an ISA and contain specific instructions to accelerate certain tasks of a computer, like AES data encryption, virtualization and vector mathematics.

Some better known examples of microarchitectures are Intel i386, Intel Nahelem, Intel Haswell, AMD K8, AMD Bulldozer and AMD Zen.

[2] *Central processing unit.*
[3] *Instruction set architecture.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Instruction_set_architecture`.
[4] *Machine code.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Machine_code`.
[5] *Microarchitecture.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Microarchitecture`.
[6] *Microcode.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Microcode`.

6.4 Threads and cores

A thread is a chain of instructions that is to be executed by the CPU.[7] Each thread is generated by a process, which can loosely be described as an instance of a computer program. A single core of a CPU can execute one thread at a time, but using a technique called time slicing and the concept of context switching,[8] can handle multiple threads concurrently.

Multithreading (software) allows a single process to spawn a multitude of threads, dividing the workload of that process. Performance benefits (can) arise when these threads are executed in parallel on multiple cores of a CPU.

The specifications of a CPU may contain references to the number of threads it "has", which should be interpreted as the maximum amount of threads it can "execute" at the same time. The fact that a single CPU core can only execute one thread at a time doesn't change, even if the CPU specifications state that it has more (twice) threads then cores. This has to do with a hardware based technique called simultaneous multithreading,[9] which will be discussed later.

6.5 Cache

Quick access to data is critical for the performance of a CPU, making data flow and storage a mayor aspect of a CPU. The main memory of a computer has a relatively high latency and low bandwidth compared to the needs of modern CPU cores, which is where cache comes into play.[10]

Cache is storage subsystem of the CPU and acts like a data buffer. It contains, among other things, copies of the data that the CPU (or process) "predicts" it will access often or in the near future, reducing the loading time of that data. The amount of cache placed on the CPU is relatively small, typically about 1:1.000, compared to the amount of main memory placed in a computer. A "cache-miss" refers to the situation where data is requested, but not stored in cache, resulting in a much longer loading time. Avoiding cache-misses is a large part of software (and hardware) optimization and can lead to very substantial performance improvements.

Cache memory pressure (memory nearly full) and the ever increasing speed of CPU cores lead to the development of multiple levels of cache. This layered structure has the advantage that it can address both the memory pressure problem as well as the demand for faster data access, without resulting in prohibitive costs. The upper most layer of cache, L1, has gotten significantly faster over time, but did not really increase much in capacity. The lowest level of cache, typically L3, saw the highest increase in capacity, but is also substantially slower then L1.

The development of multi-core processors and several levels of cache created an additional task for cache; inter-core data communication. Each core has its own private parts of L1 and L2 cache, whereas L3 cache is shared between the cores. This last level of cache is the place where data can be shared between the threads running on the different cores.

[7] *Thread (computing)*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Thread_(computing)`.

[8] *Context switch*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Context_switch`.

[9] *Simultaneous multithreading*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Simultaneous_multithreading`.

[10] *CPU cache*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/CPU_cache`.

Cache memory is a lot faster, and generally superior on many fronts, compared to main memory, because cache is made from SRAM and main memory is made from DRAM. SRAM stands for "Static Random Access Memory", whereas DRAM stands for "Dynamic Random Access Memory". Each SRAM memory cell requires 6 transistors to store a bit, whereas DRAM requires only one transistor (and a small capacitor) per bit. The downside of DRAM is that the capacitors in DRAM memory need to be recharged frequently, causing delays and other problems. This constant refreshing of the stored data gave rise to the name "Dynamic", while "Static" was used for SRAM, because it doesn't need to be refreshed. The extra hardware complexity of SRAM allows it to be much faster than DRAM, but the extra cost and space requirements on the die of the CPU also make it much more expensive.

6.6 Execution units

Execution units of a CPU core are the parts that execute the machine instructions derived from the thread running on the core. There are many different types of execution units in modern CPU cores, each with their own specific function. Notable examples of execution units are; arithmetic logic unit,[11] address generation unit[12] and floating-point unit.[13] Discussing the functions and operations of all these execution units is beyond the scope of this text, which will focus on the floating-point execution unit.

Superscalar

CPU cores have many execution units, most also have multiple execution units of the same type. Keeping all the execution units busy at the same time requires multiple instructions to be dispatched (one instruction per execution unit) simultaneously. The ability of a CPU core to dispatch multiple instructions simultaneously is called being superscalar,[14] whereas CPU's that can only dispatch a single instruction are called scalar. Superscalar capabilities are a form of instruction-level parallelism.[15]

SIMD

SIMD[16] stands for single instruction, multiple data and is a form of data level parallelism.[17] SIMD is a vector processing technique, allowing an execution unit to perform the same instruction on multiple data entries (grouped in 1D arrays called vectors) in a single clock cycle. The maximum achievable throughput increases substantially by using SIMD, but does requires all the data to be manipulated in the same way, making it less versatile.

SIMD works by exposing deep registers to execution units, containing the data vectors.[18] The instruction that the execution unit receives is performed on the complete register.

[11] *Arithmetic logic unit.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Arithmetic_logic_unit.

[12] *Address generation unit.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Address_generation_unit.

[13] *Floating-point unit.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Floating-point_unit.

[14] *Superscalar processor.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Superscalar_processor.

[15] *Instruction-level parallelism.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Instruction-level_parallelism.

[16] *SIMD.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/SIMD.

[17] *Data parallelism.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Data_parallelism.

[18] *Processor register.* Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Processor_register.

FMA3 and advanced vector extensions

The floating-point execution units found in modern Intel CPU cores are based on FMA3 (from Haswell on wards). These FMA based units are capable of three different operations:

- $a = a \cdot c + b$
- $a = b \cdot a + c$
- $a = b \cdot c + a$

The Skylake-SP uarch contains two AVX-512 execution units, with 512 bit deep registers. Each AVX-512 unit contains 8 FMA3 sub-units for "double" floating-point numbers and 16 FMA3 sub-units for "single" floating-point numbers. These AVX-512 execution units form the hardware layer of the AVX-512 ISA extension.[19]

The Haswell and Broadwell uarch contain two AVX2 execution units, with 256 bit deep registers. Each AVX2 unit contains 4 FMA3 sub-units for "double" floating-point numbers and 8 FMA3 sub-units for "single" floating-point numbers. These AVX2 execution units form the hardware layer of the AVX2 ISA extension.[20]

AMD's latest (at the time of writing) Zen architecture also supports AVX2 instructions (not AVX-512), but the hardware based implementation is completely different. It doesn't have native support for 256 bit deep registers and each AVX2 instruction takes 2 clock cycles to complete, compared to one clock cycle of Intel based AVX2 capable CPU's.

6.7 Simultaneous multithreading

Simultaneous multithreading, also known as Hyper-Threading,[21] is a technique aimed at increasing the utilization of a CPU core. Each physical CPU core represents multiple logical CPU cores, fully transparent to the Operating System. Every logical CPU core gets their own thread assignment by the OS, meaning that a single CPU core is tasked with multiple threads. The amount of logical cores per physical core differs from microarchitecture to microarchitecture. The most common is two logical cores per physical core, but Intel Xeon Phi[22] and IBM POWER9[23] based designs have 4 and up to 8 logical cores per physical core.

When a thread (assigned to a CPU core) is unable to assign tasks to every execution unit of the CPU core, instructions of a different thread assigned to the same CPU core can be dispatched to the unused execution units. One of the most common culprits for a thread to under utilize the execution units, is cache misses. SMT can be very helpful in hiding the latency caused by data requests that require multiple clock cycles to fulfill.

The performance improvements generated by SMT vary wildly with applications, from a factor of two down too a performance decrease. SMT has the largest positive effect in situations where cache-misses are frequent, instruction level parallelism per thread is low and the workloads of the threads are very heterogeneous. Math routines provided by highly optimized libraries, such as the Intel Math Kernel Library,[24] generally don't fall into this category, making SMT less beneficial for HPC purposes.

[19] *AVX-512*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/AVX-512`.
[20] *Advanced Vector Extensions 2*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#Advanced_Vector_Extensions_2`.
[21] *Hyper-threading*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Hyper-threading`.
[22] *Xeon Phi*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Xeon_Phi`.
[23] *POWER9*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/POWER9`.
[24] *Math Kernel Library*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Math_Kernel_Library`.

6.8 Operating frequency

The (high level) building blocks of digital circuits are called logic gates, which are hardware implementations of boolean operations. These logic gates are combined in intricate ways to provide more high level functionality, like the FMA operation on floating-point data. Synchronization of the logic gates is key for their operation, which is where the clock signal comes into play.[25]

The clock signal of the CPU core is a square wave signal, switching between high and low (logical "on" and "off"). The rising and falling edges of this square wave are the "timing" signals for the logic gates to evaluate their input. The frequency of this timing signal is called the operating frequency or clock frequency of the CPU core. The operating frequency of a CPU core is directly linked to the throughput of micro-operations,[26] making it a key factor in the performance of a CPU.

Turbo frequencies

Almost all modern CPU's employ some sort of dynamic operating frequency control, allowing the operating frequency of various parts of the CPU to go up or down in conjunction with demand and thermal headroom. Dynamic scaling of the operating frequencies took a flight when mobile devices became more popular, requiring momentary high performance and long battery life. The basic idea behind these techniques is that a relatively high operating frequency can be achieved for a short duration of time. This increases performance for workloads that can be completed within the time frame of the elevated operating frequency, but doesn't significantly increase the overall heat production and power consumption of the CPU. The turbo boost technology of modern CPU's is too complex to discuss in great detail in this text, but a few important aspects pertaining to floating point performance will be explained.

Specifications of the turbo frequencies are very important to the performance of a CPU, but are often reduced to a single number, masking the complete story for marketing purposes. Turbo frequencies scale down according to the workload type (normal, AVX2 and AVX512) and the amount of active cores. AVX512 workloads produce the most heat and highest power consumption because their execution units contain the most transistors, AVX2 execution units require less power and normal (non floating-point) workloads even less than that.

A more detailed specification of the turbo frequencies of an Intel Xeon Gold 6132 CPU will be provided as an example. Intel ark based information on the Xeon Gold 6132 specifies a base frequency of 2.6 GHz and a maximum turbo boost of 3.7 GHz.[27] WikiChip provides more details on the frequencies section of the Xeon Gold 6132.[28] The highest floating-point performance of the Xeon Gold 6132 is achieved when all cores are utilizing their AVX-512 execution units. The maximum turbo frequency of this workload is 2.3 GHz, which is about 40% lower than the maximum frequency (3.7 GHz) provided by Intel Ark.

[25] *Clock signal.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Clock_signal`.
[26] *Micro-operation.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Micro-operation`.
[27] *Intel Xeon Gold 6132 Processor.* Intel. 2018. URL: `https://ark.intel.com/products/123541/Intel-Xeon-Gold-6132-Processor-19_25M-Cache-2_60-GHz`.
[28] *Xeon Gold 6132 - Intel.* WikiChip. 2018. URL: `https://en.wikichip.org/wiki/intel/xeon_gold/6132`.

Chapter 7

CPU interfaces

As stated earlier, modern CPU's are more akin to SoC's than traditional processors. One of the most significant advancements has been the integration of interfaces onto the CPU package. This allows for higher bandwidth and lower latencies because fewer signals have to pass over PCB traces of the motherboard.

7.1 Memory controller

The memory controller is the connecting element between the CPU cores and the DRAM memory. Many scientific computing workloads have data sets that don't fit in the cache of the CPU, forcing extensive usage of DRAM memory. These kinds of workloads usually benefit heavily from high bandwidth and low latency main memory, making the memory controller crucial for performance. Modern memory controllers are very complex pieces of engineering and explaining their operation is well beyond the scope of this text, which will focus on their features instead.

One of the most distinguishing aspects of a memory controller is the amount of memory channels it supports.[1] A memory channel is a 64-bit wide interface to a cluster of DRAM chips, usually located on a DIMM.[2] The peak bandwidth can be increased by allowing parallel access to multiple memory channels, making dual channel memory twice as fast as single channel memory, while latency remains unaffected. Typical consumer PC and laptop CPU's have an integrated memory controller capable of dual channel, whereas the Intel Skylake-SP chips contain two memory controller, each supporting triple channel for an effective 6 channel memory system.

A second important aspect of a memory controller/system is support for Error Correcting Code, which is a technique to detect and correct certain memory errors.[3] The information stored in a memory cell can get corrupted by a faulty power supply or interaction with solar radiation, resulting in a "bit flip". ECC capable memory stores additional bits of parity data to detect and (when possible) repair these corruptions. Bit flips are not very common and most consumer applications don't suffer terribly when they encounter one (system may crash), but bit flips in sensitive, long running and expensive simulations are much more problematic. That is why ECC is almost always employed in servers, despite its drawbacks (higher cost and latency).

[1] *Multi-channel memory architecture.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Multi-channel_memory_architecture`.
[2] *DIMM.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/DIMM`.
[3] *ECC memory.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/ECC_memory`.

7.2 PCI-express

PCI-express is the dominant interconnecting interface for computer components.[4] Basically everything other then DRAM is connected to the CPU via (a derived form of) PCIe, making it a very important element of a computer. Some examples of components that are connected to the CPU via PCIe are:

- GPU
- NIC
- SAS controller
- NVMe storage

PCIe is a serial bus, introduced to replace PCI(-X)[5] and AGP.[6] The PCIe standard has seen multiple revisions (backwards compatible) over the years since its introduction, improving (among other things); bandwidth, power delivery, error correcting overhead and features. The most common implementation of the standard (at the time of writing) is version 3.x, which will be the version that this text considers.

PCIe links consist of "lanes", each lane having four physical connections. Two connections for a differential read signal and the other two for a differential write signal, amounting to a full-duplex connection. A PCIe link to a device may be a grouping of multiple lanes, ranging from one to 32 lanes per link (x1, x2, x4, x8, x16, x32). GPU's are commonly connected using a x16 link, SAS controllers and NVMe storage typically use a x4/x8 link and single port NIC's have a x1 link.

The bandwidth of a PCIe v3 x1 link is specified using Giga Transfers per second (GT/s), which specifies the amount of bits that can be transferred from the host to the client or vice versa. The PCIe v3 standard uses 128b/130b encoding for error correcting purposes, meaning that for every 130 bits transmitted, only 128 bits contain data and the remaining two bits contain a form of parity data. This means that a PCIe v3 x1 link of 8 GT/s has a bandwidth of 985 MB/s (8000 x (128/130) x (1/8) = 984.62) and a x16 link has a bandwidth of 15.75 GB/s.

DMI

The Direct Media Interface interconnect is an Intel specific protocol used to connect the CPU to the Platform Controller Hub,[7] which is (among other things) responsible for USB and SATA connectivity. DMI is a prime example of an interconnect specification derived from PCIe, with a DMI 3.0 link being nearly equivalent to PCIe v3 x4 link.

[4] *PCI Express.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/PCI_Express`.
[5] *Conventional PCI.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Conventional_PCI`.
[6] *Accelerated Graphics Port.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Accelerated_Graphics_Port`.
[7] *Platform Controller Hub.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Platform_Controller_Hub`.

QPI and UPI

Most high-end servers allow the placement of multiple identical CPU's on the same motherboard, allowing two or four CPU's to be part of the same computer. Intel QuickPath Interconnect[8] and its successor, Intel UltraPath Interconnect,[9] are interfaces primarily used for inter CPU communication on these multi socket machines.

The bandwidth of the connection between the CPU's can be important in a number of scenarios, for example:

- Memory bandwidth starved single threaded applications.
- Threads running on CPU-0 that need to communicate with a GPU connected to CPU-1.
- Results from threads running on CPU-0 and CPU-1 that need to be combined in a single thread.

The total bandwidth of a QPI/UPI connection is its transfer speed specification times four. The Intel Xeon Gold 6132 has a UPI link speed of 10.6 GT/s, amounting to 42.4 GB/s of bandwidth. Note that this is considerably less then the maximum memory bandwidth of the Xeon Gold 6132, which is 119.21 GiB/s.

[8] *Intel QuickPath Interconnect.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect`.

[9] *Intel UltraPath Interconnect.* Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Intel_UltraPath_Interconnect`.

Chapter 8

Master thesis

The master thesis project will set out to investigate the performance of a selection of algorithms and workloads, implemented on the GPU, that are common to the MEFD group. This investigation will contain an experimental part, but will focus as much as possible on the theoretical aspects that dictate the behaviour. These theoretical aspects range from elements that can be traced back to the algorithm in question, like non-parallelizable parts of the workload or communications bottlenecks, but also to aspects that are directly related to the hardware on which the workload is executed, such as the various forms of latency and memory bandwidth.

8.1 Topics

- Describe the workings of a GPU and discuss the important differences between a CPU and a GPU.
- Define a selection of test cases taken from common tasks encountered by the MEFD group and explain their relevance.
- Discuss the workings of the algorithms of the test cases with a focus on parallelizability, compute and memory operations.
- Run the test cases on the CPU and evaluate the results.
- Run the test cases on the GPU and evaluate the results.

The internship has provided a lot of background information on the workings of computers, but has focused on the CPU and its related components/subsystems. The GPU, essential to GPGPU, has not been treated in detail. The CPU was discussed first for a number of reasons, but the most important one is that it is easier to understand the workings of a GPU when you understand the workings of a CPU. The first order of business for the master thesis project is to discuss the workings of a GPU and how it is different from a CPU in relationship to HPC workloads.

The second topic of the master thesis project is to make a selection of algorithms and workloads to investigate the performance of. They must be general enough to be broadly applicable, but also need to be specific to the problems encountered by the MEFD group. It is also important to make the selection such that the amount of overlap between the workloads is minimal.

Once the test cases have been chosen, their workings need to be examined for aspects like parallelizability, compute and memory operations. These aspects form the basis for their respective performance projections.

The absolute performance of a GPU based implementation of an algorithm is not really meaningful without something to compare it to. The CPU based implementations will function as a baseline for the results of the GPU based implementations.

8.2 Challenges

- Find the correct test cases.
- Find or make software implementations of the test cases that provide a fair representation of the performance on the CPU and GPU.
- Model the (parallel) scaling results in the framework of the modified Amdahl's law.
- Map the double precision requirements of the test cases.

The usability of the results of the master thesis project hinges on the quality of the test cases. The challenge is to understand the problems of the stakeholders and translate them to test cases that mimic their problems, but also reduce them to their bare essentials.

The second challenge related to the test cases is the quality of their implementations. Comparing an unoptimized version of a CPU implementation to an unoptimized version of a GPU implementation will only provide garbage results. Both the GPU and CPU versions of the test cases need to attain a certain threshold of quality in order for them to be useful. This does not mean that they need to be heavily optimized, but they do need to approach the levels of performance of production code.

The compensated Amdahl's law forms a nice framework for modelling the parallel scalability results of the test cases. The challenge will be to relate the fitted parameters of the compensated Amdahl's law to the workings of the algorithm in question. If it is found that a test case has a communications bottleneck, how can that be traced back to the algorithm?

One of the appeals of GPGPU is that it leverages the computational horsepower of hardware that was designed for graphics related workloads of the computer. This hardware can be found in almost every computer and is generally very cheap, but does have some limitations compared to dedicated professional GPGPU components. The most significant drawback, pertaining to the workloads of the MEFD group, of consumer hardware is the lackluster double precision performance. It would be useful to investigate the relevance of reduced double precision performance in order to predict the results of the test cases on consumer level hardware.

Bibliography

[1] *Accelerated Graphics Port*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Accelerated_Graphics_Port.

[2] *Address generation unit*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Address_generation_unit.

[3] *Advanced Vector Extensions 2*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#Advanced_Vector_Extensions_2.

[4] *Amdahl's law*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Amdahl%27s_law.

[5] *Arithmetic logic unit*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Arithmetic_logic_unit.

[6] *AVX-512*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/AVX-512.

[7] *Bandwidth (computing)*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Bandwidth_(computing).

[8] *Build and run Docker containers leveraging NVIDIA GPUs*. NVIDIA. 2018. URL: https://github.com/NVIDIA/nvidia-docker.

[9] *Central processing unit*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Central_processing_unit.

[10] *Clock signal*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Clock_signal.

[11] *Context switch*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Context_switch.

[12] *Conventional PCI*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Conventional_PCI.

[13] *CPU cache*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/CPU_cache.

[14] *Data parallelism*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Data_parallelism.

[15] *Department of Mechanical Engineering, Multiscale Engineering Fluid Dynamics*. TU/e. 2018. URL: https://www.tue.nl/en/university/departments/mechanical-engineering/research/research-groups/multiscale-engineering-fluid-dynamics/.

[16] *DIMM*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/DIMM.

[17] *Double-precision floating-point format*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

[18] *ECC memory*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/ECC_memory.

[19] *Embarrassingly parallel*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Embarrassingly_parallel.

[20] *Floating-point unit*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Floating-point_unit.

[21] *Half-precision floating-point format*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Half-precision_floating-point_format.

[22] *Hyper-threading*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Hyper-threading.

[23] *Instruction set architecture*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Instruction_set_architecture.

[24] *Instruction-level parallelism*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Instruction-level_parallelism.

[25] *Intel QuickPath Interconnect*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect.

[26] *Intel UltraPath Interconnect*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Intel_UltraPath_Interconnect.

[27] *Intel Xeon Gold 6132 Processor*. Intel. 2018. URL: https://ark.intel.com/products/123541/Intel-Xeon-Gold-6132-Processor-19_25M-Cache-2_60-GHz.

[28] *Latency (engineering)*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Latency_(engineering).

[29] *Machine code*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Machine_code.

[30] *Massively parallel*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Massively_parallel.

[31] *Math Kernel Library*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Math_Kernel_Library.

[32] *Micro-operation*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Micro-operation.

[33] *Microarchitecture*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Microarchitecture.

[34] *Microcode*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Microcode.

[35] *Multi-channel memory architecture*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Multi-channel_memory_architecture.

[36] *NVIDIA CUDA Installation Guide for Linux*. NVIDIA. 2018. URL: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.

[37] *Parallel slowdown*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Parallel_slowdown.

[38] *PCI Express*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/PCI_Express.

[39] *Platform Controller Hub*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Platform_Controller_Hub.

[40] *POWER9*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/POWER9.

[41] *Processor register*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Processor_register.

[42] *SIMD*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/SIMD.

[43] *Simultaneous multithreading*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Simultaneous_multithreading.

[44] *Single-precision floating-point format*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Single-precision_floating-point_format.

[45] *Superscalar processor*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Superscalar_processor.

[46] *Thread (computing)*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Thread_(computing).

[47] *Xeon Gold 6132 - Intel*. WikiChip. 2018. URL: https://en.wikichip.org/wiki/intel/xeon_gold/6132.

[48] *Xeon Phi*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Xeon_Phi`.

[49] W.M. Zuberek and T.D.P. Perera. ‘Performance Analysis of Distributed Iterative Linear Solvers’. In: *7th WSEAS Int. Conf. on Mathematical Methods and Computational Techniques in Electrical Engineering* (), pp. 194–199.

Appendix A

CUDA installation

The following information is to be considered complementary to the documentation that NVIDIA provides. Users who don't have a whole lot of experience with the CUDA toolkit, or are otherwise unfamiliar with the installation process, might find useful information.

Regarding the linux section, users on distributions other than Ubuntu, might be of the opinion that the following text is biased towards Ubuntu. The recommendations that follow are based on my personal experience, and as such may very well be biased.

Linux

Installing CUDA on linux can be cumbersome, if you don't stick to the officially supported distributions. These vary according to the CUDA toolkit version you wish to install. A list of supported linux distributions for CUDA 9.1 can be found at the system requirements section of the CUDA installation guide.[1] The following is a list of linux distributions that NVIDIA supports on AMD64 capable hardware:

- CUDA 9.2 supports Fedora 27, OpenSUSE 42.3, SLES 12 SP3, RHEL/CentOS 7 and 6, Ubuntu 16.04 LTS and 17.10.
- CUDA 8.0 supports Fedora 23, OpenSUSE 13.2, SLES 12 and 11 SP4, RHEL/CentOS 7 and 6, Ubuntu 16.04 LTS and 14.04 LTS.
- CUDA 7.5 supports Fedora 21, OpenSUSE 13.2, SLES 12 and 11 SP3, RHEL/CentOS 7 and 6, Ubuntu 15.04 and 14.04 LTS.
- CUDA 6.0 supports Fedora 19, OpenSUSE 13.2, SLES 11 SP3 and SP2, RHEL/CentOS 6 and 5, Ubuntu 13.04 and 12.04 LTS.

Another possibility is to use a prepackaged version of CUDA, generally provided by the linux distribution itself. Debian 8 (jessie) is a good example of a linux distribution that is not officially supported by NVIDIA, but does provide a prepackaged CUDA toolkit in the backports repository.

Even if a prepackaged CUDA toolkit is available for your distribution, it might be better to use a version provided by NVIDIA. Ubuntu 16.04 LTS is an officially supported distribution for a number of CUDA toolkit versions, but also provides a prepackaged version of CUDA 7.5. It might be tempting to just install the prepackaged version (`'sudo apt install nvidia-cuda-toolkit'`), but the way in which certain libraries are provided deviates from the official NVIDIA way, which may cause problems later on.

[1] *NVIDIA CUDA Installation Guide for Linux.* NVIDIA. 2018. URL: `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`.

Windows

Installing CUDA on Windows is relatively trouble free. Just download the version you wish to install from NVIDIA and follow the instructions of the installer.

The installer also includes a version of the NVIDIA driver, but it might be outdated. I would recommend to install the latest driver that NVIDIA provides for your GPU.

Docker

NVIDIA Docker is a relatively new development that aims to facilitate running CUDA applications and development in a containerized world. It can also be used to setup a CUDA development environment with relative ease. (only supported on linux)

GPGPU acceleration in a containerized environment has the inherent problem that containerized environments are by design hardware agnostic, limiting access to the GPU. NVIDIA provides software for Docker that circumvents this problem, called `‘nvidia-docker2‘`. The host operating system requires a supported NVIDIA graphics driver, Docker and the aforementioned `‘nvidia-docker2‘`, The CUDA toolkit and other user space software runs inside the container. A list of prerequisites and a quick installation guide are available at the wiki of the projects github page.[2]

One of the prerequisites is a CUDA capable device of an architecture newer than Fermi, meaning: Kepler, Maxwell, Pascal and Volta (at the time of writing). Even though CUDA 8.0 and earlier development environments are available using NVIDIA Docker, targeting Fermi and Tesla will not be possible using NVIDIA Docker.

NVIDIA docker Installation instructions:

```
# Install docker (provided by Canonical) and curl
sudo apt install docker.io curl

# Add the NVIDIA docker repository to the system
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
distribution=$(.  /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list
| sudo tee /etc/apt/sources.list.d/nvidia-docker.list
sudo apt update

# Install nvidia-docker2 and reload the Docker daemon configuration
sudo apt install nvidia-docker2
sudo pkill -SIGHUP dockerd

# Add the user to the docker group (logoff and login to take effect)
sudo usermod -aG docker "username"

# Test NVIDIA docker with nvidia-smi
docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

[2] *Build and run Docker containers leveraging NVIDIA GPUs.* NVIDIA. 2018. URL: https://github.com/NVIDIA/nvidia-docker.

Example

As an example, the following command will open an interactive shell in a CUDA 8.0 development environment Docker container:

```
docker run -it --runtime=nvidia --rm nvidia/cuda:8.0-devel /bin/bash
```

Running ‘`nvcc -V`‘ and ‘`gcc --version`‘ in this bash prompt results in:

```
nvcc:  NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2016 NVIDIA Corporation
Built on Tue_Jan_10_13:22:03_CST_2017
Cuda compilation tools, release 8.0, V8.0.61

gcc (Ubuntu 5.4.0-6ubuntu1 16.04.9) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

As can be seen from these results, NVIDIA made the decision to base their Docker container on Ubuntu 16.04 LTS.

Toolchain

A reliable and well documented toolchain is paramount to software development. Achieving this doesn't have to be difficult, but it does limit the options somewhat.

Compiler

An important aspect of the prerequisites of the CUDA toolkit is the supported c/c++ compiler. This also varies from version to version of CUDA and can be found in the same table that also provides the officially supported linux distributions.

Linux

Choosing the right linux distribution is an important step, because it determines the (system default) compiler, libraries and available support.

GCC support of CUDA:

- CUDA 9.2 supports gcc up-to 7.x
- CUDA 8.0 supports gcc up-to 5.x
- CUDA 7.5 supports gcc up-to 4.x
- CUDA 6.0 supports gcc up-to 4.7

You could install (or compile) a different version of gcc on your distribution, but you would also need to install or recompile various libraries to ensure ABI compatibility.

```
sudo apt install gcc-4.9 g++-4.9

sudo ln -s /usr/bin/gcc-4.9 /usr/local/cuda/bin/gcc
sudo ln -s /usr/bin/g++-4.9 /usr/local/cuda/bin/g++
```

The above commands install gcc 4.9 on Ubuntu 16.04 LTS and links that to the installed CUDA toolkit. This (and similar) solutions have been reported to work correctly, but your results may vary.

Ubuntu and derivatives

Ubuntu 16.04 LTS is a good choice for linux development in general and CUDA development as well. The system default compiler is gcc 5.4, which is officially supported by the CUDA toolkit versions 8.0 and 9.x (possibly later versions too). The availability of the Hardware Enablement (HWE) stack provides the choice between a rock solid linux 4.4 LTS kernel and a more recent kernel for those who need or want it.

Ubuntu 16.04 LTS is an officially supported linux distribution for the versions 8.x and 9.x. NVIDIA does not officially support Ubuntu 16.04 for CUDA 7.5, but as mentioned earlier, a prepackaged version of CUDA 7.5 is available in the official Ubuntu repositories.

An other advantage of the 16.04 LTS release is that it is supported until sometime in 2021, and even longer support is available for paying customers. Not having to reevaluate the toolchain and development environment every year is a big plus.

Derivatives of Ubuntu LTS releases, like for example Kubuntu 16.04, are similar enough to cause minimal difficulties. The only real exception to this are window managers (kwin, mutter, compton) and their interaction with NVIDIA proprietary drivers. Certain third party software is known to produce more segmentation faults if the graphics stack is not completely standard, and a different window manager can cause problems.

Windows

The situation on Windows is similar, as the latest (point) release of the Visual C++ compiler is not always compatible. Every CUDA release that officially supports Windows 10, also supports the Visual C++14.0 (Visual Studio 2015) compiler, making it a good candidate.

CUDA toolkit version

If your project doesn't require the latest and greatest features of the CUDA toolkit, it might be advantageous to stick to an earlier version of CUDA, for compatibility sake. Version 8.0 has hardware support for the Fermi, Kepler, Maxwell and Pascal architectures (newer versions removed Fermi support in favor of Volta and older versions lack Pascal support).

NVIDIA graphics driver on linux

Installing the proprietary NVIDIA graphics driver on linux can be a hassle, requiring kernel header files, dkms and compiler. A further complication is that certain installation methods require the manual blacklisting of the open source and unofficial NVIDIA driver (nouveau). Various linux distributions have made a lot of progress over the last couple of years in aiding the user in the installation of the NVIDIA graphics driver, but the installation of the CUDA toolkit also installs the driver that NVIDIA recommends (this behavior can be altered).

My personal recommendation would be to stick to the defaults that NVIDIA provide. The installation instructions included in the CUDA toolkit documentation are easy to follow and tell the user how to prepare the system for the CUDA toolkit as well as the included graphics driver.

A last note on the graphics driver is that the CUDA toolkit requires a minimum version, which depends on the version of the toolkit. The by NVIDIA included driver will always meet that requirement, but if you decide to manually force a different version of the graphics driver, you will have to take this into account.