Optimizing matrix calculations using NVIDIA graphics cards

Bachelorproject - BIBAPRO1PE

Dagrún Eir Ásgeirsdóttir & Markus Æbelø Faurbjerg
daas@itu.dk — mfau@itu.dk

Supervised by:
Peter Sestoft - sestoft@itu.dk

February 7, 2024

Abstract

In this bachelor project, we investigate the potential performance increase of using the CUDA programming language on NVIDIA Graphics Processing Units (GPU) compared to Central Processing Unit (CPU) based sequential implementations in the programming language C. We do this for matrix addition, multiplication and Lower-Upper Decomposition (LUD). Our primary focus points for optimization is parallalization using multiple threads, and use of shared memory.

Our findings show that the CUDA-based GPU implementations can result in significant performance improvements for all three types of calculations by utilizing parallelization. While the usage of shared memory also showed improvement in performance for multiplication, and potentially LUD, its impact varies across different tasks, based on memory access patterns and frequency. In our LUD implementation, we also highlight how optimal performance is achieved by utilizing the combined strengths of both the sequential processing capabilities of a CPU and the parallel processing power of a GPU. We have analysed our results with the help of NVIDIA Nsight Compute, and identified further optimization opportunities. We calculate some estimated performance increases achievable from these opportunities.

We conclude that there are performance gains achievable by utilizing GPUs, when compared to basic sequential CPU implementations. This is particularly the case for tasks that can be parallelized. We show this with a combination of our own results and insight from NVIDIA Nsight Compute.

Furthermore, our results show that when implementing solutions that include both sequential and parallelizable parts, using the strengths of CPUs and GPUs in combination is favourable, as per the CUDA programming model.

Contents

1 Introduction

This is a Bachelor Project by Dagrún Eir Ásgeirsdóttir & Markus Æbelø Faurbjerg. It is for the Software Development programme at the IT-University in Copenhagen. It is produced in 2023 and is 15 ECTS. The project is named 'Optimizing matrix calculations using NVIDIA graphics cards' and is supervised by Peter Sestoft.

1.1 Motivation

Computational processing power has increased massively in the last decades. Currently, the advent of artificial intelligence is increasing the demand for higher processing power. The ability to handle massive parallel workloads is especially attractive.

The Core Processing Units in average consumer hardware is not well suited for parallel work. Different architectures have been developed to take on the task of performing parallel workloads efficiently. The potential for many-fold performance increases makes graphics processing units an interesting and important tool. For this project, we have decided to investigate ways to utilize graphics processing units to increase the performance of matrix operations.

1.2 Preliminary knowledge

A basic understanding of the C programming language or an equivalent language will be needed to understand the CUDA kernels. The features specific to CUDA programming will be explained in section 2.3.1.

1.3 Problem definition

We will investigate whether performance improvements can be achieved in matrix calculations by using CUDA on a graphics processing unit (GPU) in an NVIDIA graphics card instead of a central processing unit (CPU). The matrix calculations we will perform include addition, multiplication, and lower-upper decomposition.

Initially, we will create a basis for comparing basic CPU and GPU implementations by implementing sequential CPU- and GPU-based solutions. The CPU implementation will be done using the C programming language, and the GPU implementation will be done using CUDA. Next, we will attempt to optimize the GPU implementation based on theoretical considerations of the advantages of using graphics cards, including insights from the 'CUDA C++ Programming Guide' created by NVIDIA. Using the benchmarking principles from 'Microbenchmarks in Java and C#' (Sestoft), we will conclude if a performance improvement has been achieved by using the CUDA architecture.

1.4 Method

To understand what performance can be expected from a simple solution, all operations will first be implemented in C and converted to a CUDA equivalent implementation. To compare the two, the implementations will be timed. Once a baseline performance has been established, the operations will be optimized to utilize the strengths of the CUDA architecture. To further investigate the strength of the implementations, additional software will be used to analyze the performance of the CUDA implementations.

2 Background

In this chapter, we will go over knowledge needed to understand this project. The chapter will include what a matrix is and relevant matrix operations. It will also introduce graphics processing units, the CUDA architecture and how we benchmark our results.

2.1 Matrix

A matrix is a rectangular table of numbers stored in rows and columns. Each number in the table is called an element or an entry. As illustrated in figure 1, each element can be identified by a combination of its row and column position.

$$\begin{array}{c|cccc|} & 1 & 2 & \dots & j \\ \hline 1 & a_{11} & a_{12} & \dots & a_{1j} \\ 2 & a_{21} & a_{22} & \dots & a_{2j} \\ \dots & \dots & \dots & \dots & \dots \\ i & a_{i1} & a_{i2} & \dots & a_{ij} \\ \hline \end{array}$$

Figure 1: The red number represents an element's row position and the blue number represents its column position.

A number of operations can be performed on matrices. The relevant operations will be explained in the remainder of this section (2.1). In order to do these operations, it is necessary that the matrices are compatible. The requirements for compatibility differ depending on the operation.

2.1.1 Matrix addition

To add two matrices, every element of matrix A is added with the element on the same position in matrix B, such that the element at position c_{ij} in the resulting matrix C is equal to $a_{\text{ij}} + b_{\text{ij}}$.

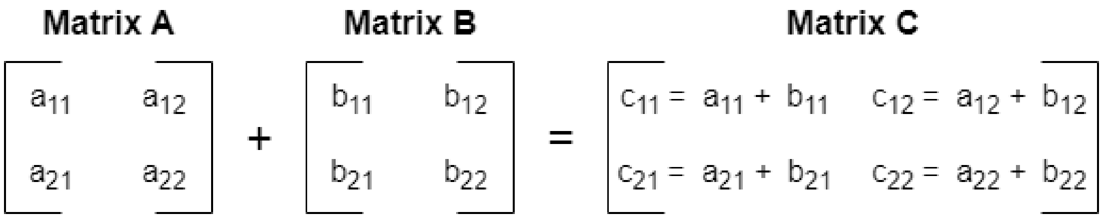

Matrix A
Matrix B
Matrix C
a_{11} a_{12} a_{21} a_{22}
+
b_{11} b_{12} b_{21} b_{22}
=
$c_{11} = a_{11} + b_{11}$
$c_{12} = a_{12} + b_{12}$
$c_{21} = a_{21} + b_{21}$
$c_{22} = a_{22} + b_{22}$

Figure 2: The elements a_{22} and b_{22} are added to calculate the value of element c_{22}.

For two matrices to be compatible for addition, they must have the same number of rows and the same number of columns. That is $matrixA.rows = matrixB.rows$ and $matrixA.cols = matrixB.cols$.

2.1.2 Matrix multiplication

For two matrices to be compatible for multiplication, the number of columns in matrix A must be equal to the number of rows in matrix B. As illustrated in figure 3 this means that a matrix A with the dimensions 3 × 2 and a matrix B with the dimensions 2 × 3 will produce matrix C with the dimensions 2 × 2.

To calculate the value of element C_{mn} in the resulting matrix C, we multiply elements from matrix A and matrix B pairwise, like this:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

When all elements have been multiplied pairwise, the result of each element multiplication is summed. The result of the summation is placed in matrix C.

In figure 2.1.2, the element at c_{11} is calculated based on the elements in matrix A and matrix B, which are highlighted.

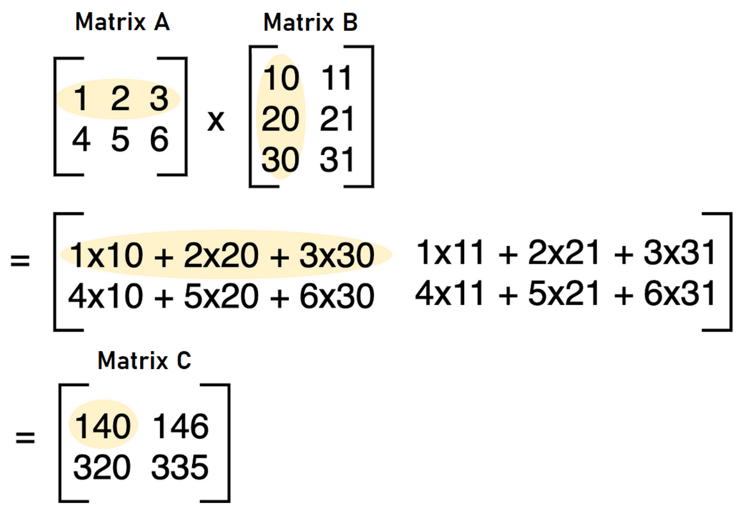


Figure 3: Multiplying matrix A and matrix B results in matrix C, which has the same number of rows as matrix A and the same number of columns as matrix B.

2.1.3 Lower-Upper decomposition

Matrix decomposition splits a matrix into parts, that when multiplied together, recreate the original matrix. This can, for instance, be used to simplify computations. For this project, we have chosen to attempt to optimize Lower-upper decomposition.

A Lower-Upper Decomposition (LUD) is a technique used to decompose a matrix **A** into two triangular matrices: a Lower Triangular Matrix **L** and a Upper Triangular Matrix **U**. This decomposition is represented as:

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

- **L**, being a lower triangular matrix, contains non-zero elements only on the diagonal and below, with zeros above the diagonal
- **U**, being a upper triangular matrix, contains non-zero elements only on the diagonal and above, with zeros below the diagonal

There are multiple methods of LUD. Figure 4 illustrates Doolittle's method. One distinct feature of this method is storing 1s on the diagonal of the lower triangular matrix. For this project, all implementations use this method as a foundation.

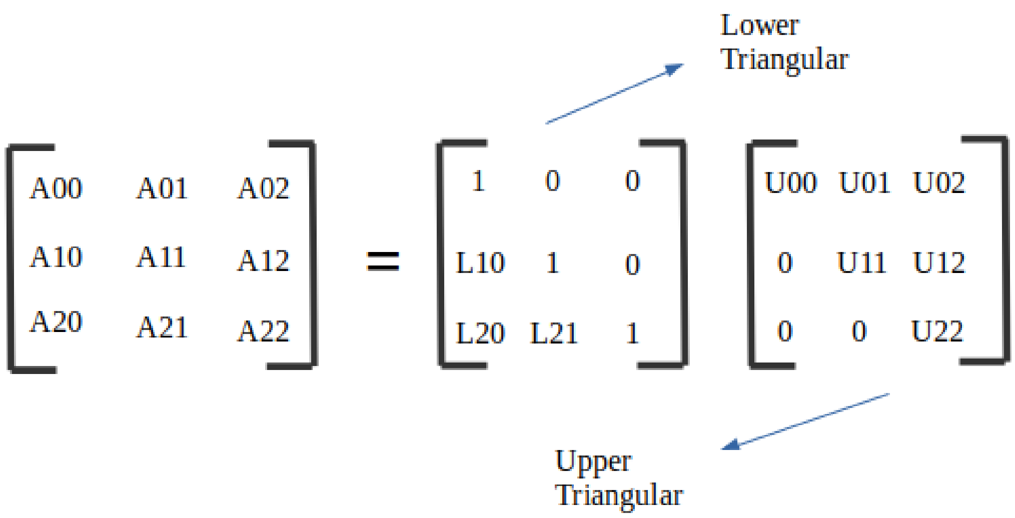


Figure 4: Lower-upper decomposition following Doolittle's method from the GeeksForGeeks website[1].

The benefits of performing LUD on a matrix include:

- **Efficiency:** Allows for cheaper computations when solving linear equations or performing matrix inversion.
- **Numerical Stability:** Enhances numerical stability, when also implementing pivoting, as it helps avoiding problems connected to division by small numbers or zero.

Similar benefits can be achieved by other methods of matrix decompositions and different methods are best suited for different applications. LUD is a comparably straight forward decomposition method and especially useful in numerical linear algebra.

Pivoting

Implementing pivoting means that during the decomposition, rows and/or columns are swapped to ensure the best possible results when performing division. This would most likely mean swapping so that the dividing element is as large as possible. Full pivoting includes swapping both rows and columns, while partial pivoting topically only involves swapping rows. In most cases partial pivoting is considered to be sufficient and stable [2].

When applying partial pivoting, the decomposition looks as follows:

$$\mathbf{P} \cdot \mathbf{A} = \mathbf{L} \cdot \mathbf{U}$$

P being the permutation matrix, which is an identity matrix with the rows exchanged in the same manner as A. When multiplying A with the permutation matrix P, the result is the permuted version of A, which can then be decomposed into L and U.

When applying full pivoting, the decomposition looks as follows:

$$\mathbf{P} \cdot \mathbf{A} \cdot \mathbf{Q} = \mathbf{L} \cdot \mathbf{U}$$

Where P, A, L and U remain the same, and where Q is the permutation matrix where the columns are exchanged in the same manner as A.

2.2 Graphics Processing Unit

2.2.1 What is a Graphics Processing Unit

Within computing, a Graphics Processing Unit (GPU) is a specialized hardware component located on a computers graphics card, made to excel at parallel computation[3]. To understand the role of a GPU, it helps to compare it with a Central Processing Unit (CPU).

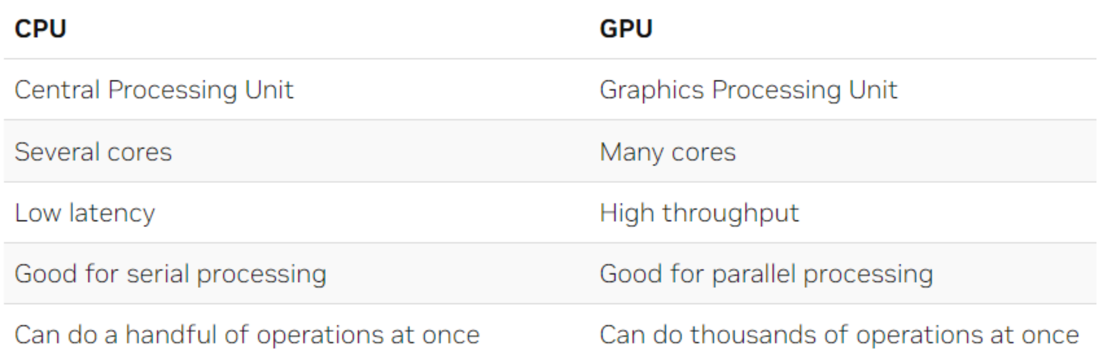
A computer needs to be able to handle diverse tasks like running the operating system, programs and retrieving data from a hard drive. Since a CPU is responsible for handling these tasks it needs to be a versatile component.

While a CPU is a crucial part of running a system, it can be impractically slow in certain contexts. A CPU is not designed to excel at tasks that are most optimally performed by splitting it into numerous subtasks. This could be tasks like deep learning, artificial intelligence or computing the graphics for a 3D environment in a game.

Unless explicitly stated, all information in section 2.2 is sourced from The CUDA C++ Programming Guide[4] (henceforth cited as *the CUDA guide*).

The CUDA guide explains the different purposes of CPUs and GPUs. CPUs are designed to excel at computing a sequence of operations, called a *thread*. It is designed to execute a few tens of threads in parallel. A GPU has individually slower threads, but amortizes its slower single-thread performance by executing thousands of threads in parallel, which allows it to achieve greater throughput.

The table in figure 5 from NVIDIA's website [5] summarizes the main differences between CPUs and GPUs. In the following chapter, we will explain the physical architecture of the GPU and how it differs from the architecture of a CPU. This will highlight how the hardware is designed for parallel processing.



CPU	GPU
Central Processing Unit	Graphics Processing Unit
Several cores	Many cores
Low latency	High throughput
Good for serial processing	Good for parallel processing
Can do a handful of operations at once	Can do thousands of operations at once

Figure 5: The main differences between a CPU and a GPU[5].

2.2.2 Graphics Processing Unit hardware

The design of a GPU allocates a greater proportion of transistors to data processing at the expense of cache and flow control. This can be seen in figure 6 from the CUDA guide. It illustrates the architectural differences between CPUs and GPUs.

The GPU architecture has many more cores (represented by green rectangles) than caches and control units (purple and yellow rectangles). This design introduces more memory access latency, but a GPU can effectively mask it by compensating with higher computation strength. Since cache and flow control are more expensive in terms of transistors, GPUs can deliver much higher instruction throughput and memory bandwidth at a similar price to a CPU.

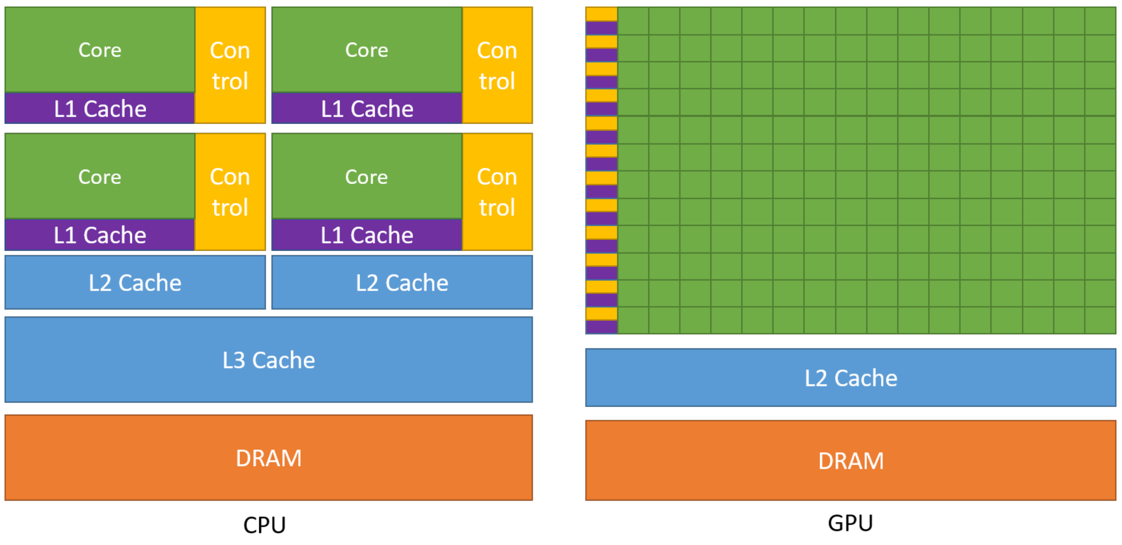


Figure 6: The architectural differences between a CPU and a GPU[4]. The GPU has much smaller caches and control flow units, compared to the CPU. The GPU also has many more and simpler cores for each of its caches and control units.

Threads and the Thread Hierarchy

A thread is a sequential unit of execution that operates independently and can operate concurrently with other threads. Threads are essential to parallel systems such as a GPU, and enable simultaneous execution of multiple tasks, greatly improving both performance and efficiency.

Threads are often grouped together, to better manage and coordinate their executions. On NVIDIA GPUs, threads are grouped into warps of a fixed size, typically containing 32 threads. All threads within the same warp execute the same instruction in parallel on different data, synchronize with each other and can share memory. This also means that threads within the same warp cannot perform different tasks simultaneously, due to the Single Instruction, Multiple Threads (SIMT) model, which will be explained further in a following section.

Instead of being bound to the small fixed size of thread warps, threads can be grouped into larger *thread blocks*. The size of a thread block should be a multiple of the warp size of the given device, because threads in a warp are indivisible and must be processed together. The block size is specified by the program launching the threads and can therefore be sized according to the requirements of the task at hand. To execute the threads, they are scheduled on Streaming Multiprocessors (SMs), which contain a collection of cores that execute the thread warps concurrently.

A GPU contains multiple SMs and each of these manages a set number of thread warps. The number of thread warps per SM varies between different GPU architectures. The SM also contains the memory associated with these thread warps, known as *shared memory*, which is a fast, low-latency, on-chip memory. This will be explained further in the following section. When choosing block sizes, the number of thread warps should therefore not exceed the number contained within a single SM, as warps on different SMs would not be able to access the same shared memory, which counteracts the idea of a thread block.

By grouping threads into blocks, the same conditions as explained for warps apply, meaning that all threads within a block run concurrently, synchronize with each other and share memory. This also means that threads within the same block are not able to perform separate tasks simultaneously. Thread blocks can be arranged in one, two or three dimensions, allowing them to be customized even further to fit different use cases. All threads therefore have a 3-component vector called *threadIdx*, that holds x, y and z values, so that the thread can be identified in up to three dimensions.

A recent addition to the thread hierarchy in CUDA are *thread block clusters*. They were introduced with the NVIDIA Compute Capability 9.0, along with the new NVIDIA Hopper architecture, which was released in September 2022. A thread block cluster allows for even broader control and enhanced memory sharing. Grouping thread blocks into clusters, guarantees the blocks to be launched simultaneously by the *GPU processing cluster*, in the same manner that the SM launches all threads within a thread block. This is not achievable without thread block clusters, as thread blocks don't have access to each others shared memory, and cannot be guaranteed to launch simultaneously, as they might be run on different SMs.

The outermost layer of the thread hierarchy is the *grid*, which contains multiple thread blocks or thread block clusters. Thread blocks are grouped in a grid to be able to launch several blocks in parallel, and the grid size is therefore determined by the use case. The relationship between thread blocks, thread block clusters and grids can be seen in figure 7.

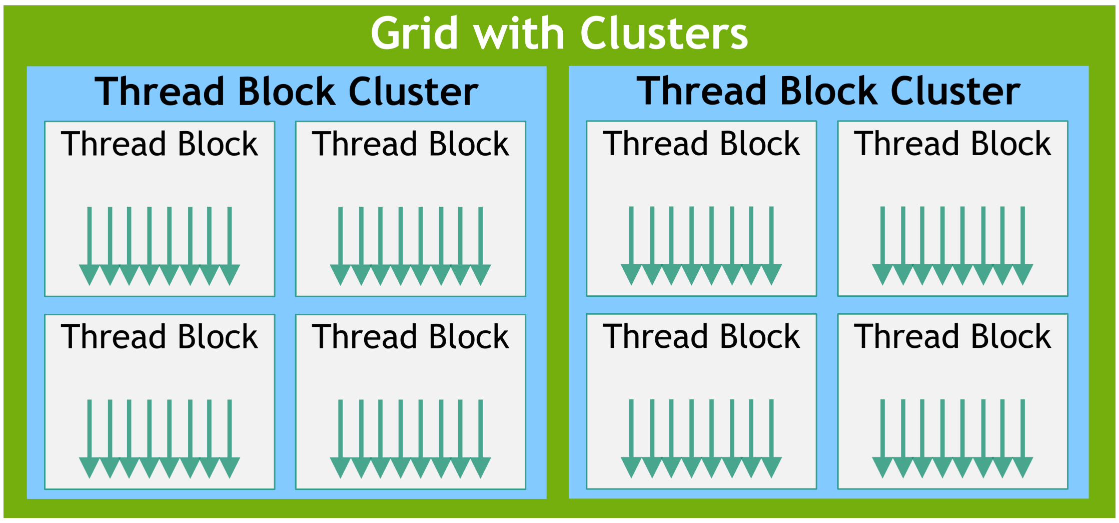


Figure 7: The thread hierarchy within a GPU with CUDA architecture. Threads are grouped into thread blocks. Multiple thread blocks can be managed in thread block clusters, that allow them to share memory[4].

Memory hierarchy

As briefly mentioned in section 2.2.2, a GPU possesses different types of memory, accessible by different parts of the thread hierarchy.

Each thread has its own set of *registers*. These are only accessible by the thread itself and are very fast, but also have a very limited quantity. If more registers are needed than available, excess data will spill into *local memory* instead, which is considerably slower. It would instead be better to use the shared memory when the registers are not sufficient.

Threads within a block have access to shared memory. The shared memory is also fast and limited in size, but do not spill like registers do. Shared memory is managed by the programmer, and is typically used for storing data frequently accessed or exchanged by threads within the same block. Trying to allocate more memory than the available shared memory can lead to issues. The CUDA program may fail to launch, encounter runtime errors, or performance may be negatively impacted. With the introduction of thread block clusters, blocks within a cluster, gained access to a *distributed shared memory*, allowing them to access each others shared memory. This is not possible on previous CC versions, where thread blocks do not have access to each others shared memory.

The grid has the largest memory space, called *global memory*. This memory is accessible by all threads and grids, as well as the host CPU, and is therefore very versatile. It is however the slowest, and therefore not preferable to use in most cases. Data should instead be loaded from global memory into shared memory or even registers depending on the use case. It does however have its uses, as it could sometimes be preferable to have data accessible to multiple grids. When using global memory, one would however need to keep in mind that race conditions could occur when threads from different blocks attempt to write to the same location, making the data incorrect. When using shared memory, race conditions can be prevented by synchronizing the threads, but no such method exists for threads on different blocks.

The global memory resides in the device memory, meaning the DRAM attached to the GPU. This memory is referred to as off-chip because, even though being physically attached to the GPU, it is not located on the chip itself. In contrast to the off-chip memory, the on-chip memory is located directly on the GPU chip. On-chip memory includes registers, shared memory, the L1 and L2 caches.

The L1 and L2 caches are integral parts of the on-chip memory. An L1 cache is associated with each SM and temporarily holds data, that is frequently accessed by the threads managed by the SM. The L2 cache serves a similar purpose, but is shared across all SMs. Both L1 and L2 caches are slower than registers and shared memory, but are faster than the global memory. Furthermore, the L1 cache is faster than the L2. Therefore, the L1 cache is the preferable memory to use when the registers and the shared memory are not sufficient. If it is not possible to use the L1 or L2 cache, the global memory can be used.

Newer NVIDIA GPUs with a Compute Capability (CC, to be discussed in section 2.3) of 7.x or higher, have a unified data cache for shared memory and L1. This means that after allocating shared memory, the remaining data cache (or all, if no shared memory is used) is used for L1 cache. This means that data access to the shared memory and L1 are equally fast. Using shared memory allows for control over the memory, but makes it only accessible by a single thread block, making it more preferable in cases where the programmer wants to manually control the memory, but otherwise not needed. Different CC versions allow for the shared memory capacity to be set to different sizes, but never the entire data cache, as at least 28KB need to remain for the L1 cache[4].

All caches are managed automatically and their content is determined by previous memory accesses. They can therefore not be managed by the programmer in the way that shared memory can.

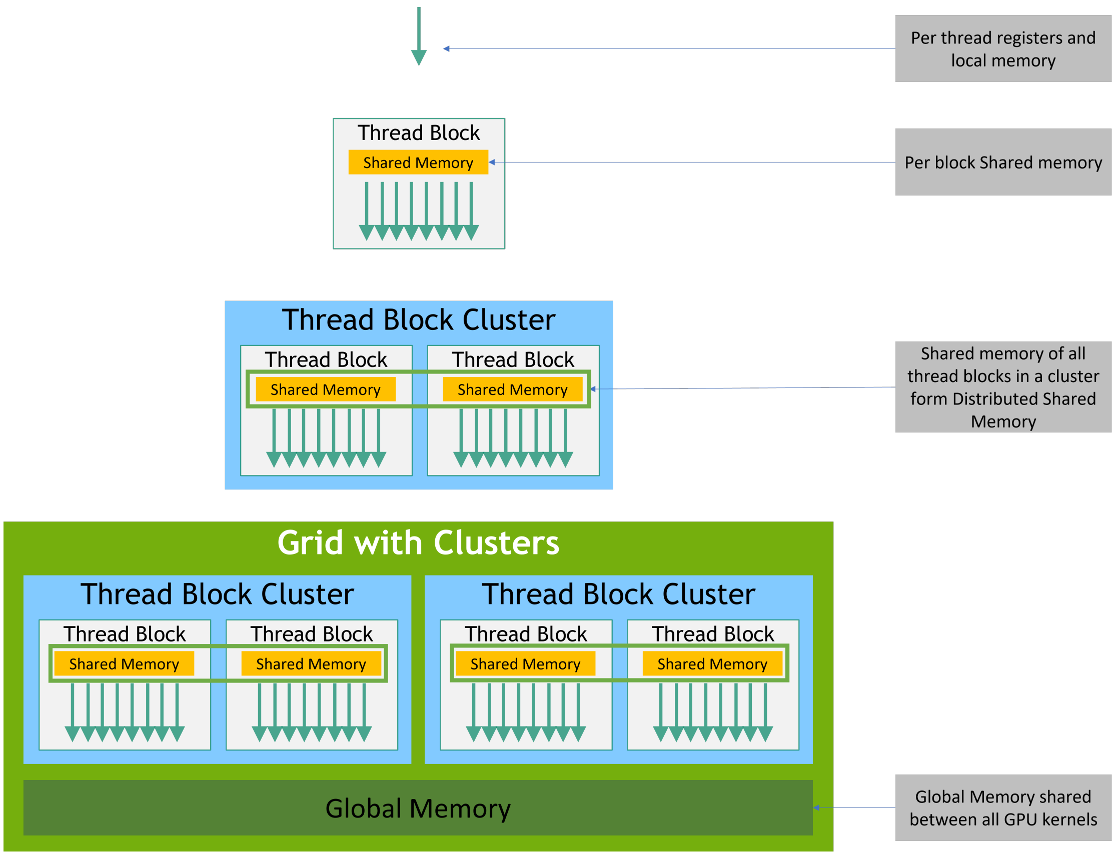

Per thread registers and local memory
Thread Block
Shared Memory
Per block Shared memory
Thread Block Cluster
Thread Block
Shared Memory
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Thread Block Cluster
Global Memory
Global Memory shared between all GPU kernels

Figure 8: The memory hierarchy within a GPU with CUDA architecture[4].

SIMT Architecture

The Streaming Multiprocessors are responsible for scheduling and executing multiple threads concurrently. This is done with the help of the Single-Instruction, Multiple-Thread architecture (SIMT). SMs schedule threads by dividing a block, or multiple blocks, into the predetermined thread warps. Each warp then executes one instruction at a time, as warps cannot process multiple different instructions simultaneously.

If a warp is used to perform operations that require fewer threads than the warp contains, the remaining threads will idle. For example, imagine that a warp with 32 threads execute line 7-8 in figure 9. If only 14 threads satisfy the *if* statements on line 7, the remaining 18 threads in the warp will idle while the code on line 8 is executed. Once an operation is completed, this process repeats for subsequent operations. This is referred to as *warp-divergence*. In general, warp-divergence happens when conditional statements result in multiple branch paths. Each of these branch paths are executed separately and threads on other branch paths are disabled [4].

The SIMT architecture is what makes a GPU efficient at computing tasks that can be broken into subtasks and computed in parallel. One such task is matrix calculations, where individual elements in the resulting matrix can be computed concurrently. In comparison, a CPU is optimized to excel at performing more complex and general-purpose tasks. While CPUs also support parallelism via multiple cores and the Single-Instruction, Multiple-Data (SIMD) architecture, they do so on a smaller scale.

2.3 CUDA

CUDA is a parallel programming model designed to leverage the increasing number of processing cores in computing hardware. It works exclusively with NVIDIA GPUs. The first version of the CUDA (CUDA 1.0) Software Development Kit (SDK) was released in 2006. As of November 2023, the current version is 12.3 [6].

As well as more sophisticated software, GPUs have been through many versions of hardware design architectures. Newer architectures have brought higher computational performance and new computation capabilities. These incremental improvements are categorized by CUDA Compute Capability (CC). The current highest CC available is 9.0 for the NVIDIA H100 Tensor Core GPU [7]. The CC versions of the GPUs that we will be using are listed in section 3.1.

2.3.1 Programming with CUDA

Chapter five of the CUDA Guide explains the CUDA programming model. We will go over the fundamentals here, so that we can understand how the CUDA code works. This will also allow us to reason about how the programmatic choices we make impacts the performance of our programs.

CUDA comes with a software environment that allows C to be used as a high-level programming language. This means that a CUDA program can manage when to use the CPU and when to use the GPU. This allows the programmer to execute serial computation on the CPU and parallel computation on the GPU.

The parallel computation is defined inside Kernels. A kernel is a C function that is executed N times in parallel by N CUDA threads. A kernel is defined by using the *__global__* declaration specifier.

The number of CUDA threads that will execute a call to a kernel is decided by using the <<< ... >>> *execution configuration syntax*. The <<< ... >>> syntax defines the number of threads per thread block and the number of thread blocks per grid. This can be specified using the types *int* or *dim*3, which is an integer vector type that can take up to three arguments. Any components that are not specified are automatically initialized with the value 1.

Each of the threads in a thread block will be given a *thread ID*. Each thread ID is accessible in the kernel by using built-in variables. These are the *ThreadIdx* variables mentioned in section 2.2.2. They are referenced individually using *ThreadIdx.x*, *ThreadIdx.y* and *ThreadIdx.z*. If the block is one-dimensional, you will only need ThreadIdx.x. If it is two-dimensional, you will need ThreadIdx.x and ThreadIdx.y. If it is three-dimensional, you will need all three.

There are also built-in variables to access the block index and the block dimension. These variables are called *blockIdx* and *blockDim*. In combination, the variables can be used to access a specific thread, in a specific block in the grid.

The example kernel and kernel invocation in figure 9 illustrates the concepts from this section. The code in the *__global__ void MatSub()* kernel will be compiled to device code and executed on the GPU, while the main function will be compiled to host code and be executed on the CPU.

The values of the variables i and j on line 4 and 5 are calculated by using the values of the block index, block dimension and thread index variables of this specifi1c instance of the kernel. This means that the index of a kernel will designate which element in the matrix it will access. Since it is a two dimensional block, it uses the *blockDim.x* and *blockDim.y* variables.

The dim3 variables on line 14 and 15 are given as arguments to the execution configuration syntax on line 16. The *blockDim* variable defines the dimensions of the block to be 16×16. The value of the *gridDim* variable is the result of the calculation performed on line 15. This means that the amount of blocks used will increase with the size of N.

```
 1  // Kernel definition
 2  __global__ void MatSub(float A[N][N], float B[N][N], float C[N][N])
 3  {
 4      int i = blockIdx.x * blockDim.x + threadIdx.x;
 5      int j = blockIdx.y * blockDim.y + threadIdx.y;
 6
 7      if (i < N && j < N)
 8          C[i][j] = A[i][j] - B[i][j];
 9  }
10
11  int main()
12  {
13      ...
14      dim3 blockDim(16, 16);
15      dim3 gridDim((N + blockDim.x - 1) / blockDim.x,
16                   (N + blockDim.y - 1) / blockDim.y);
17      MatSub<<<gridDim, blockDim>>>(A, B, C);
18      ...
19  }
```

Figure 9: Example code snippet that uses the fundamental building blocks of the CUDA programming model.

2.3.2 Optimization strategies for CUDA based graphics cards

The CUDA C++ Best Practices Guide [8] describes ways to optimize CUDA applications. In chapter 5, the guide describes three fundamental ways to get started. Only the third will be relevant for this project. The first way is to use parallel libraries. One example of a parallel library is cuBLAS, which is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime [9].

The second is to use parallelizing compilers. This often means using directives-based approaches, where the programmer adds notation in the code that helps the compiler pinpoint specific sequential sections, that are candidates for parallelization.

The third way is to code to expose parallelism. This approach is more manual, in the sense that it is the programmer that writes the parallel kernels. The guide describes this approach as the most straightforward one, when the majority of the running time of the application is spent in a few relatively isolated portions of the code. That lines up with the way we will work, as there will be a supporting environment around the matrix operations, in order to set up matrices, manage memory and run the kernels a specified number of times. This is the approach we will use for this project, as it will allow us to directly define implementations and compare their performance.

2.4 NVIDIA Nsight Compute

NVIDIA Nsight Compute (NNC) is a profiler tool that can be used with CUDA. It can provide varying levels of performance metrics by running CUDA kernels. Inspecting these metrics can help discover bottlenecks in performance. One important metric is the GPU throughput, which measures the rate at which the GPU processes data. NNC itself also provides suggestions for optimizations along with an estimation of the potential performance increase.

NNC was released in 2018 and does not have full backwards compatibility with older GPU architectures.

2.5 Benchmarking

2.5.1 Measurement strategy

The measurement strategy we use is based on Microbenchmarks in Java and C#[10]. The paper is a written guideline for doing benchmarking on managed execution platforms, namely Java and C#.

Managed execution platforms use runtime environments like the Java Virtual Machine and the Common Language Runtime for C#. This is because Java and C# are high-level programming languages, that come with features like automatic memory management.

While C and CUDA are low-level languages that do not use managed execution platforms, there are still many guidelines that are worthwhile for this project. Below is a list of guidelines that we deemed valuable for our project and have chosen to follow.

Clock time

We will be using wall-clock time measured on the relevant system to track the performance of implementations, as we will be running tests across both C and CUDA. This is, to compare kernels that are exclusively handled by the GPU, we will use the GPU to time them. When comparing C code, we will use the CPU for timing. When we are testing roundtrip code that starts execution at the CPU, transfers data to the GPU and receives data back, we will use the CPU.

Minimize background processes

We measure absolute time spent, not only the CPU time spent. This means that it is important to minimize the processes running in the background on the operating system. If tests are run with different amounts of external workloads on the system, the performance would be affected and the results would lose precision or even be misleading.

Run the program from a command line

Furthermore, the program should not be running within an Integrated Development Environment. These environments can interfere with the performance through injecting debugging code, trying to supervise the execution, or simply consuming processing power by running the software. To remove any such interference, the program should exclusively be launched using a command line tool.

Automate multiple runs

To reduce the time spent performing tests, we will create a program that allows testing of multiple functions. This program will also execute each function multiple times, to reduce the impact of artifacts on single executions.

2.5.2 Measurement implementations

To time our CUDA kernels, we follow the the CUDA Guide. The guide describes how to time applications in the 'Performance Metrics' chapter. The code in figure 10 is taken from the guide. It illustrates how to measure the running time of a kernel.

It uses *cudaEventRecord()* immediately before the kernel execution on line 7 and immediately after on line 10.

The *cudaEventElapsedTime* function returns the elapsed time in milliseconds as a floating-point value. The resolution of the timing is approximately half a microsecond.

We follow the approach in figure 10, with a modification that increases the readability of our test environment. We separate the code into the three functions *createTimer()*, *beginTimer()* and *endTimer()*. This removes implementation details like defining the variables *cudaEvent_t start, stop* from the test environment.

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS);

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Figure 10: Source code to time the execution of a CUDA kernel from 'How to time code using CUDA events' from the CUDA Guide.

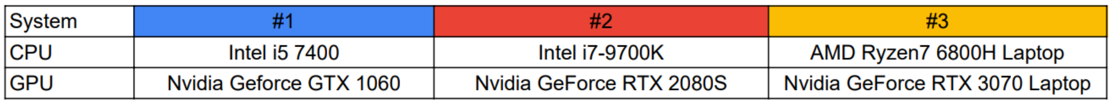
To perform measurements on the CPU in C we use Windows-specific API functions. Like with the CUDA timer, we implemented the three methods *createTimer()*, *beginTimer()* and *endTimer()*.

There are a number of alternatives to this, like the C library function clock_t clock(). We decided to go with the Windows API functions as they offered the most precision and all test systems run on Windows.

The measurement implementations are documented in the appendix. The implementation for CUDA is in section 10.2 and the implementation for C is in section 10.3.

3 Test systems

We will be using three test systems to measure the performance of our code. Using three, as opposed to just a single system, will highlight how using different versions of Compute Capability impacts performance. This will become relevant in section 4.2.4 when the GTX 1060 GPU in system #1 will perform significantly worse, due to its older Compute Capability version. The CPUs and GPUs in the three systems can be seen below in figure 11.



System	#1	#2	#3
CPU	Intel i5 7400	Intel i7-9700K	AMD Ryzen7 6800H Laptop
GPU	Nvidia Geforce GTX 1060	Nvidia GeForce RTX 2080S	Nvidia GeForce RTX 3070 Laptop

Figure 11: The CPUs and GPUs in the three test systems.

Going forward, we will refer to the CPUs as i5, i7 and AMD and the GPUs as 1060, 2080 and 3070.

3.1 Hardware used for testing

Figure 12 shows technical specifications of the three GPUs. The cards span three release cycles of NVIDIA GPUs. NVIDIA uses a naming convention for the Geforce line that identifies a GPUs generation and it's performance, relative to its generation.

The last two digits of the number represents the relative performance of the card. For most generations, NVIDIA will release cards with the digits __50, __60, __70, __80 and __90, where __50 is the card with the lowest performance and __90 is the card with the highest performance. The first one or two digits represents the generation. The GPUs in our test systems cover the 1000, 2000 and 3000 series.

The technical specifications of the CPUs are listed in figure 13. Their performance is of less importance. They serve the purpose of showing how sequential solutions do not scale well with GPUs. As the CPUs are designed to perform well sequentially, they will outperform the GPUs when working sequentially, but will be unable to match the performance of GPUs when moving from sequential program to parallel programs.

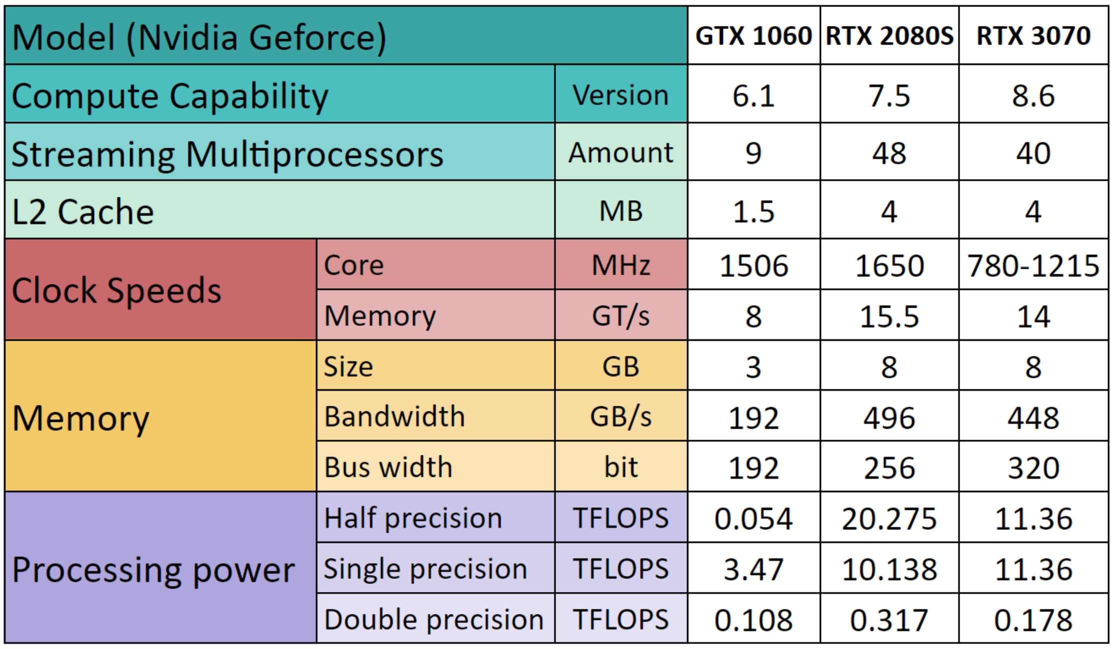
3.2 Expected Performance

The processing power we work with is quantified in terms of FLOPS. The acronym stands for Floating Point Operations per Second. An example of a Floating point operation (FLOP) is an add operation. This means that adding two matrix elements is counted as one floating point operation. It is worth noting, however, that the GPUs can do either add, multiply or multiply-add. Using multiply-add means that two operations are performed at once.

We expect that the relative performance of the GPUs will roughly follow their listed processing power. We will be doing Single Precision (32 bit IEEE binary floating-point) calculations, meaning that the 3070 should have a slight performance advantage over the 2080, based on their listed single precision processing power. Similarly, we expect the 1060 to deliver approximately a third of the performance of the 2080 and the 3070. Furthermore, we wonder if the Compute Capabilities of the GPUs may influence the performance, since there are numerous architectural differences between versions. For example, the 1060 and 3070 has a throughput of 128 'Results per Clock Cycle per Multiprocessor' [2] for 32-bit add, multiply and multiply-add operations, while the 2080 does 64 [4].

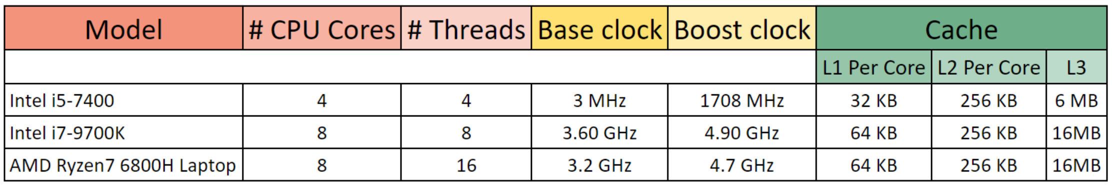
While we expect the relative performance of the cards to be mostly consistent, we expect that the theoretical performance and the observed performance of the cards will vary widely. Solutions that do not work well should significantly reduce the observed performance, while more sophisticated approaches will achieve a higher percentage of the maximum theoretical performance.

[2]The table in figure 66 in the appendix lists the throughput for the Compute Capabilities used in the three GPUs.



Model (Nvidia Geforce)			GTX 1060	RTX 2080S	RTX 3070
Compute Capability		Version	6.1	7.5	8.6
Streaming Multiprocessors		Amount	9	48	40
L2 Cache		MB	1.5	4	4
Clock Speeds	Core	MHz	1506	1650	780-1215
	Memory	GT/s	8	15.5	14
Memory	Size	GB	3	8	8
	Bandwidth	GB/s	192	496	448
	Bus width	bit	192	256	320
Processing power	Half precision	TFLOPS	0.054	20.275	11.36
	Single precision	TFLOPS	3.47	10.138	11.36
	Double precision	TFLOPS	0.108	0.317	0.178

Figure 12: Technical specifications of the GPUs in the test systems.



Model	# CPU Cores	# Threads	Base clock	Boost clock	Cache		
					L1 Per Core	L2 Per Core	L3
Intel i5-7400	4	4	3 MHz	1708 MHz	32 KB	256 KB	6 MB
Intel i7-9700K	8	8	3.60 GHz	4.90 GHz	64 KB	256 KB	16MB
AMD Ryzen7 6800H Laptop	8	16	3.2 GHz	4.7 GHz	64 KB	256 KB	16MB

Figure 13: Technical specifications of the CPUs in the test systems.

4 Implementations

In the following chapter, we will go over multiple implementations of the matrix operations and the testing environment that supports it. The source code we wrote can be found in the *source code* directory, though code will be shown when relevant. Test data is also included in the *Data* directory.

The code is running within a respective C or CUDA test environment. Our procedure to run a program in C is:

1. Create matrices (Allocate memory, initialize and populate)
2. Start timer
3. Perform matrix operation
4. End timer
5. Output collected data to file
6. Deallocate memory

Our procedure to run a program in CUDA requires a bit more setup, as the matrices have to be transferred from host memory to device memory.

1. Create matrices on host (Allocate memory, initialize and populate)
2. Allocate memory on device and copy matrices from host to device
3. Start timer
4. Perform matrix operation
5. End timer
6. Output collected data to file
7. Deallocate memory

Both architectures use a **Matrix** and a **Timer** struct. The Matrix struct defines a matrix. A matrix has rows, columns and stores its' data in a row-major fashion in memory. We populate each element in a matrix with a pseudo-random number generator. All testing is done on square matrices and each size of matrix is tested 100 times per test system.

C code is compiled with the GNU Compiler Collection. CUDA code is compiled using the NVIDIA CUDA Compiler. The commands we used are:

```
gcc -o [outputFilename] [myCode.c]
nvcc -ccbin [filepath] -o [outputFilename] [myCUDACode.cu]
```

Filepath specifies the path to the host compiler binary. This path is specific to the system setup.

4.1 Matrix addition

4.1.1 A baseline implementation of matrix addition in C

The sequential addition implementation in figure 14 computes the resulting matrix sequentially. The method takes three matrices. The matrices M1 and M2 are the input matrices and M3 is the result matrix, where the resulting elements will be stored.
By iterating over j in the inner loop on line 4, the resulting matrix C is calculated row-wise, one element at a time.

```c
 1  void sequential(Matrix M1, Matrix M2, Matrix M3)
 2  {
 3      for (int i = 0; i < M3.rows; i++)
 4      {
 5          for (int j = 0; j < M3.cols; j++)
 6          {
 7              M3.data[i * M3.cols + j] =
 8                  M1.data[i * M1.cols + j] +
 9                  M2.data[i * M2.cols + j];
10          }
11      }
12  }
```

Figure 14: The C function *sequential* for matrix addition on a CPU.

This implementation consist of two loops, making the asymptotic running time $O(N \times M)$, where N and M represent the number of rows and columns in the matrices. By definition, the two matrices have the same number of rows and the same number of columns. If they did not, they would be incompatible for addition. Since every loop cycle includes one floating point operation, the amount of floating point operations required to do the matrix computation is also $O(N \times M)$.

We use square matrices of varying sizes when testing the performance. Since M will be equal to N, the amount of floating point operations needed will be N^2. We expected the running time to increase at a similar rate.

The test results in figure 15 compare the performances on the three test systems. When analyzing the observed data, we found that the empirical growth rate is close to the expected rate.

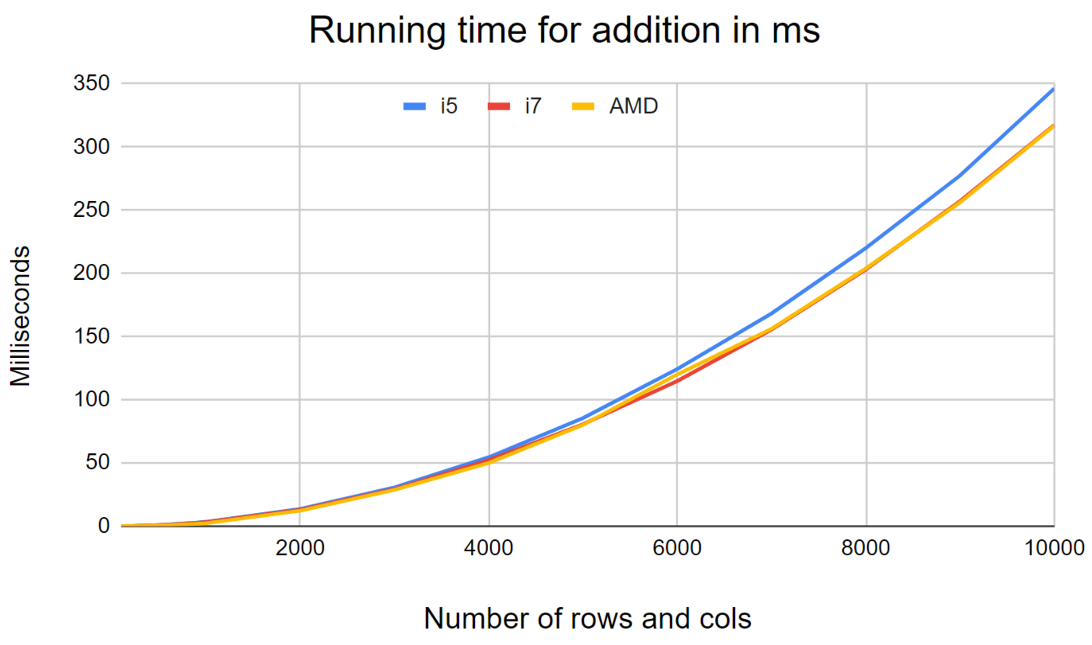


Figure 15: Running time for the three CPUs used for testing, when they do matrix addition on matrices of varying sizes.

This implementation has one outspoken advantage and drawback. The advantage is that it accesses the elements in the input matrices in a coalesced manner. That is, since the elements of the matrices are stored in memory as illustrated in figure 16, the next two elements to be added are more likely to be available in the caches, than if the element accesses were not coalesced. If the elements are not in cache, they have to be loaded in from slower forms of memory, increasing the running time.

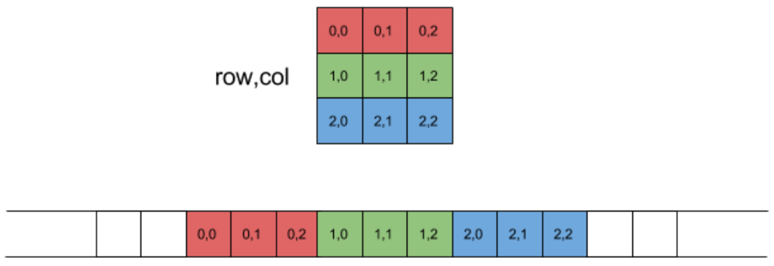


Figure 16: The elements of a matrix are stored stored in a row-major order in memory when using the C architecture.

The drawback of the implementation is its sequential nature. By performing a single operation in the inner loop, the computation utilizes a just one thread, where it could otherwise utilize multiple threads for computation (assuming there are no compiler optimizations or the like).

This is likely less detrimental on CPUs, since they have far fewer threads than GPUs. In the following chapter, we will demonstrate that a sequential implementation performs far worse on a GPU.

4.1.2 CUDA - Sequential implementation

The CUDA equivalent to a sequential matrix addition algorithm is implemented with the Sequential kernel in figure 17. There are two main differences between the implementations. The first is, that this kernel only transfers the matrix elements from the host to device. Since the device code does not have access to the struct that defines matrix structures, the elements are transferred using the CUDA memory management function `cudamemcopy`. A matrix' elements are stored in an array of floats. In the code, these arrays are named M1, M2 and M3.

The second difference is that this method also takes the two arguments *int rows* and *int cols*. Since entire matrix structures are not passed to the kernel, it does not have access to the other information, than what the *Matrix.data* array contains. The C programming language does not have a built-in method to determine the length of an array. On the device, the float arrays are just a block of memory. For this reason, we also pass rows and cols to the kernel, such that we have bounds for the loops available in the device code.

```
__global__ void Sequential(
    float *M1, float *M2, float *M3,
    int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            M3[rows * i + j] = M1[rows * i + j] + M2[rows * i + j];
        }
    }
}
```

Figure 17: The CUDA kernel *Sequential* for matrix addition on a GPU. The kernel takes five arguments. The input pointers M1 and M2 point to the matrix data, while the M3 pointer points to where the resulting matrix is stored. The rows and cols parameters define the number of rows and cols in the matrices.

Despite the two implementations being almost identical, the CUDA kernel performs significantly worse. At square matrices of size 100, the difference between the slowest CPU and the fastest

GPU is a factor of approximately 25. At size 1000, the difference between the fastest CPU and the slowest GPU is a factor of approximately 6250. This is caused by the difference in the hardware mentioned in the CPU and GPU comparison table in figure 5.

Matrix / Processor	100	200	300	400	500	600	700	800	900	1000
i5	0.03	0.12	0.39	0.57	0.90	1.26	1.70	2.21	2.78	3.44
i7	0.03	0.11	0.32	0.52	0.83	1.16	1.61	2.08	2.62	3.25
AMD	0.02	0.08	0.26	0.39	0.59	0.85	1.15	1.48	1.85	2.32
1060	2.55	30.31	122.16	353.50	909.03	1886.07	3435.84	5787.31	9533.10	14522.98
2080	0.77	5.02	20.40	61.80	153.08	317.84	579.39	996.17	1646.73	2488.13
3070	0.73	5.66	22.37	62.66	159.13	325.41	593.77	1001.68	1653.69	2489.77

Figure 18: Running time in milliseconds for sequential addition on all six processors.

It seems reasonable to assume that the sequential memory access allows for the L1 (and even L2) caches to be utilized by the GPU. This means that the memory access is fast, but will not make much of a difference for the running time, as the memory access is not what we expect to be the main reason for the slow running time.

When running the sequential implementation, a single thread will perform the entire task. Even though we launch kernels with multiple threads, according to the logic in figure 19, the threads will perform the same operations in parallel. In an ideal scenario, the 256 threads would execute different tasks each. In this case, the threads will execute the same task 256 times, providing no more performance than a single thread. This is due to the SIMT architecture.

Neither of the GPU's strengths of parallel processing and doing thousands of operations at once, is utilized at all with this approach. On the other hand, the CPU performance is not impacted like the GPU performance is. Like NVIDIA explains, a CPU excels at serial processing and only doing a handful of operations at once, which is exactly what happens with a sequential approach.

```
// Define block and grid dimensions for CUDA kernel
dim3 blockDim(16, 16);

dim3 gridDim((MCols + blockDim.x - 1) / blockDim.x,
             (MRows + blockDim.y - 1) / blockDim.y);
```

Figure 19: The code that we use to set block and grid dimensions

One might think that computing a task multiple times in parallel, is about as fast as computing the task just once. As it turns out, it is actually a lot slower than performing the task just once. Figure 20 shows the performance of the sequential kernel, when it is launched with a single thread.

Matrix / GPU	100	200	300	400	500	600	700	800	900	1000
1060	1.78	5.71	12.94	23.81	36.32	52.21	70.70	92.15	120.92	151.35
2080S	0.53	1.91	3.60	6.31	10.21	16.52	23.94	31.02	38.94	46.47
3070	0.50	1.94	3.98	6.67	10.04	20.84	30.23	40.23	50.56	62.37

Figure 20: Running time in milliseconds for sequential addition when the GPUs utilize a single thread instead of multiple threads.

Launching with a single thread is done by giving **1,1** as the arguments to the execution configuration syntax when invoking the kernel, as illustrated in figure 21.

```
Sequential<<<1,1>>>
```

Figure 21: An example of how to launch the *Sequential* kernel using a single thread.

Comparing the running time when launching the kernel with many threads and a single thread, it is clear that having many threads working on the same task concurrently, does not scale well. At matrices with 100 rows and 100 columns, the performance increases by a factor of about 1.5 when using one thread. At 1000 rows and 1000 columns, the performance increases by a factor of about 40.

Initially, we suspected that the worse performance of using many threads, was due to *bank conflicts*. Bank conflicts happens when threads from different warps, request access to addresses within the same *memory banks*. When this happens, the access to the memory is serialized. We found out, however, that this only applies to shared memory and is thus not the issue. Instead, we hypothesize that the worse performance of using many threads, is due to the much larger amount of memory accesses, which are also less likely to be coalesced, compared to a single thread approach.

To improve the running time further, we want to switch to an approach that utilizes the GPU's parallel processing capabilities. This will allow us to perform multiple different computations in parallel.

4.1.3 CUDA - Parallel implementation

The first step towards better utilizing the capabilities of the GPU architecture, is to do the computation in parallel. This is done with the Parallel kernel in figure 22.

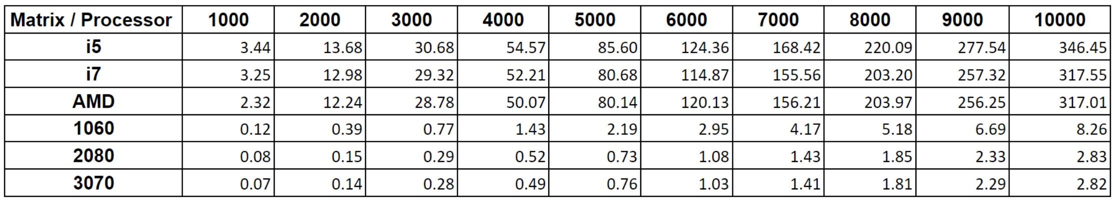
```
1  __global__ void Parallel(
2      float *M1, float *M2, float *M3,
3      int rows, int cols)
4  {
5      int row = blockIdx.y * blockDim.y + threadIdx.y;
6      int col = blockIdx.x * blockDim.x + threadIdx.x;
7
8      if (row < rows && col < cols)
9      {
10         int index = row * cols + col;
11         M3[index] = M1[index] + M2[index];
12     }
13 }
```

Figure 22: The CUDA kernel *Parallel* for matrix addition on a GPU.

The parallelization is achieved by utilizing the built-in variables of the threads running the kernel. By calculating a row- and column-index for each thread, as seen on line 5 & 6, we can use a thread to perform the calculation on the elements with the same index in the matrices. This ensures that all indices are calculated simultaneously. Since we use a two-dimensional representation of the threads, we calculate the corresponding one-dimensional index on line 10. All threads that have an index higher than the number of elements in the matrices idle. Again, this is a consequence of warps executing a set amount of threads at a time.

It is important to launch the kernel with appropriate grid and block dimensions. Using fewer threads than the number of matrix elements would return an incorrect result, as not every element would be computed. This is avoided by setting an appropriate *gridDim* on run-time in figure 19.

Changing from the Sequential kernel to the Parallel kernel increases the performance significantly. At square matrices of size 1000, the difference between the fastest CPU and the slowest GPU is a factor of approximately 19. At size 10,000, the difference between the slowest CPU and the fastest GPU is a factor of approximately 123.



Matrix / Processor	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
i5	3.44	13.68	30.68	54.57	85.60	124.36	168.42	220.09	277.54	346.45
i7	3.25	12.98	29.32	52.21	80.68	114.87	155.56	203.20	257.32	317.55
AMD	2.32	12.24	28.78	50.07	80.14	120.13	156.21	203.97	256.25	317.01
1060	0.12	0.39	0.77	1.43	2.19	2.95	4.17	5.18	6.69	8.26
2080	0.08	0.15	0.29	0.52	0.73	1.08	1.43	1.85	2.33	2.83
3070	0.07	0.14	0.28	0.49	0.76	1.03	1.41	1.81	2.29	2.82

Figure 23: Comparing the running time of the *sequential* CPU function and the *Parallel* CUDA kernel. All results are in milliseconds.

4.1.4 CUDA - Shared memory implementation

Having moved from a sequential implementation to a parallel one, the next step was to try to utilize the shared memory. Doing this, would allow for faster memory access. The question was whether or not it would improve the performance, as it would add an extra step into the execution. In the parallel implementation, elements were accessed and then used to perform operations. With a shared memory approach, elements are accessed and loaded into shared memory, used for operations and then saved back into global memory. It is required to store the results in global memory, as data in shared memory can not be transferred back to the host.

The kernel *SharedMemory* in figure 24 utilizes the same indexing logic as the Parallel kernel. It adheres to the thread index and calculates the matrix index accordingly.

```
 1  __global__ void SharedMemory(
 2      float *M1, float *M2, float *M3,
 3      int rows, int cols)
 4  {
 5      int row = blockIdx.y * blockDim.y + threadIdx.y;
 6      int col = blockIdx.x * blockDim.x + threadIdx.x;
 7
 8      __shared__ float sharedMemory1[16 * 16];
 9      __shared__ float sharedMemory2[16 * 16];
10
11      int index = row * cols + col;
12      int sharedIndex = threadIdx.y * blockDim.x + threadIdx.x;
13
14      // Ensure the index is within the matrix dimensions
15      // before loading into shared memory
16      if (index < rows * cols)
17      {
18          sharedMemory1[sharedIndex] = M1[index];
19          sharedMemory2[sharedIndex] = M2[index];
20      }
21
22      // Ensure all threads within the block have loaded data
23      __syncthreads();
24
25      if (index < rows * cols)
26      {
27          M3[index] =
28              sharedMemory1[sharedIndex] + sharedMemory2[sharedIndex];
29      }
30  }
```

Figure 24: The CUDA kernel *SharedMemory* for matrix addition on a GPU.

The shared memory arrays are declared on line 8 and 9 using the *__shared__* keyword. The array sharedMemory1 stores the element data for the M1 matrix and sharedMemory2 does the same for the M2 matrix. The data is loaded from M1 and M2 into shared memory on line 16-20, and then accessed via shared memory to calculate the result on line 27.

The column chart in figure 25 compares the performance of the two kernels. The *sharedMemory* kernel does not provide better performance than the *Parallel* kernel for addition. We tested across various sizes of matrices with similar outcomes. Initially, we expected the use of shared memory to speed up the performance, due to shared memory being the fastest memory type.

However, when performing addition, each element is accessed just once, and in the case of the

Parallel kernel, this memory access happens in global memory. When loading the data into shared memory, this global memory access happens anyway, in order to fetch the data. This explains why we did not see increased performance. If the same data had to be accessed multiple times, we would expect to see increased performance. This is because repeated memory accesses, after the initial loading, would be via shared memory instead of global memory.

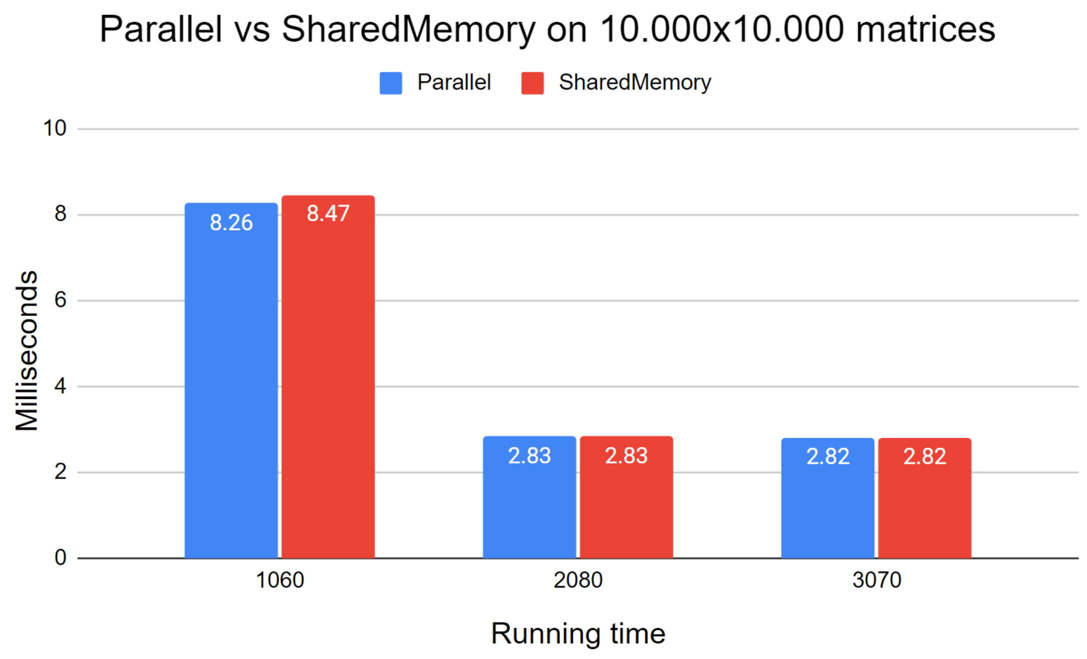


Figure 25: Comparing the running time of the *Parallel* and the *SharedMemory* kernels.

4.2 Matrix multiplication

4.2.1 A baseline implementation of matrix multiplication in C

A sequential multiplication implementation in C can be seen in figure 26. It takes the same three matrix inputs and then accesses the elements in the result matrix via a nested for loop, iterating over the rows and columns.

To calculate the result element, another loop is used to access the elements needed for the multiplication calculation, as seen in section 2.1.2. The result is added up using the **sum** variable seen on line 7, and once the result has been fully calculated, it is added to the result matrix.

```
 1  void sequential(Matrix M1, Matrix M2, Matrix M3)
 2  {
 3      for (int i = 0; i < M1.rows; i++)
 4      {
 5          for (int j = 0; j < M2.cols; j++)
 6          {
 7              float sum = 0.0f;
 8              for (int k = 0; k < M1.cols; k++)
 9              {
10                  float a = M1.data[i * M1.cols + k];
11                  float b = M2.data[k * M2.cols + j];
12                  sum = sum + (a * b);
13              }
14              M3.data[i * M3.cols + j] = sum;
15          }
16      }
17  }
```

Figure 26: The C function *sequential* for matrix multiplication on a CPU.

The theoretical running time of *multiplicationSequential()* is equal to $N{\cdot}M{\cdot}K$, where N is the number of rows in M1, M is the number of columns in M2 and K is the number of columns in M1. For square matrices where $N{=}M{=}K$, the asymptotic running time is then $O(N^3)$.

Figure 27 shows the performance of the three CPUs. As we expected, the observed performance for square matrices scale at approximately $O(N^3)$.

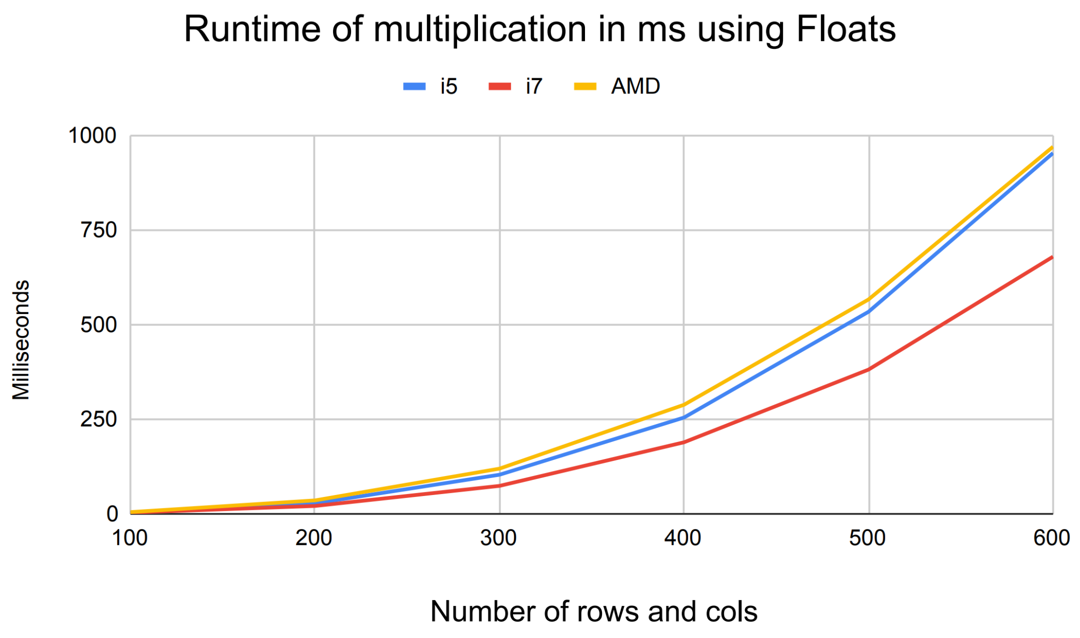


Figure 27: Running time of the *sequential* function on the three CPUs.

4.2.2 CUDA - Sequential implementation

We again start with a sequential implementation in CUDA, to have a direct comparison to the CPU implementation. The differences between the C and the CUDA implementations remain the same. The CUDA kernel only has access to the matrix element data, and it therefore also uses the constants M1Rows, M1Cols and M2Cols as loop barriers.

The three loops remain otherwise identical, and the result element is calculated and stored into the result matrix in the same manner as the C implementation.

```
__global__ void Sequential(
    float *M1, float *M2, float *M3,
    int M1Rows, int M1Cols, int M2Cols)
{
    for (int i = 0; i < M1Rows; i++)
    {
        for (int j = 0; j < M2Cols; j++)
        {
            float sum = 0.0f;
            for (int k = 0; k < M1Cols; k++)
            {
                sum += M1[i * M1Cols + k] * M2[k * M2Cols + j];
            }
            M3[i * M2Cols + j] = sum;
        }
    }
}
```

Figure 28: The CUDA kernel *sequential* for matrix multiplication on a GPU.

The table in figure 29 shows the average running time in milliseconds for each of the six processors. Again, the CUDA implementation performs worse. When multiplying square matrices of size 100x100, the slowest CPU beats the performance of the fastest GPU by a factor of about 2. At matrices of size 600x600, the fastest CPU beats the performance of the slowest GPU by a factor of about 7.

Matrix / Processor	**100**	**200**	**300**	**400**	**500**	**600**
i5	3.45	27.22	103.41	254.19	534.00	951.94
i7	3.08	23.46	86.13	204.19	429.44	689.09
AMD	4.48	35.41	120.45	283.62	555.30	963.58
1060	30.52	233.25	788.80	1839.56	3631.67	6268.19
2080	9.83	101.03	371.55	903.77	1789.06	3076.94
3070	10.13	85.07	349.39	1005.81	2031.35	3634.17

Figure 29: The running time in milliseconds for the sequential implementations on the CPUs and the GPUs.

As the CUDA Guide states, the first optimization step is to locate sections of code that can be run in parallel. After investigating how much parallelization impacts multiplication performance, we will explore more ways to potentially optimize the performance.

4.2.3 CUDA - Parallel implementation

```
__global__ void Parallel(float *M1, float *M2, float *M3)
{
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if (row < M1Rows && col < M2Cols)
        {
                float sum = 0.0f;

                for (int i = 0; i < M1Cols; i++)
                {
                   sum += M1[row * M1Cols + i] * M2[i * M2Cols + col];
                }
                M3[row * M2Cols + col] = sum;
        }
}
```

Figure 30: The CUDA kernel *Parallel* for matrix multiplication on a GPU.

The parallel implementation again shows a significant performance improvement, from utilizing the GPUs strengths discussed in section 4.1.3. At matrices of size 100x100, the slowest GPU beats the performance of the fastest CPU by a factor of about 50. At matrices of size 600x600, the fastest GPU beats the performance of the slowest CPU by a factor of about 1900. When doing multiplication on larger matrices, a GPUs ability to amortize many threads, allows it to execute much faster than a CPU.

Matrix / Processor	100	200	300	400	500	600
1060	0.06	0.11	0.26	0.69	1.35	2.27
2080	0.03	0.06	0.13	0.20	0.39	0.54
3070	0.03	0.04	0.09	0.18	0.30	0.50

Figure 31: The running time for the parallel implementation on the GPUs measured in milliseconds.

4.2.4 CUDA - Shared memory and tiling implementation

As mentioned when discussing the shared memory addition kernel, we expected to see improvements in running time when implementing a shared memory multiplication kernel instead of global memory. We expected this based on the faster memory access, when needing to access the same elements of the input matrices multiple times.

Figure 32 from the CUDA guide illustrates the memory access when calculating a single element in the result matrix. For calculating element C[i][j], the entirety of row [i] is accessed from matrix A and the entirety of col [j] from matrix B.

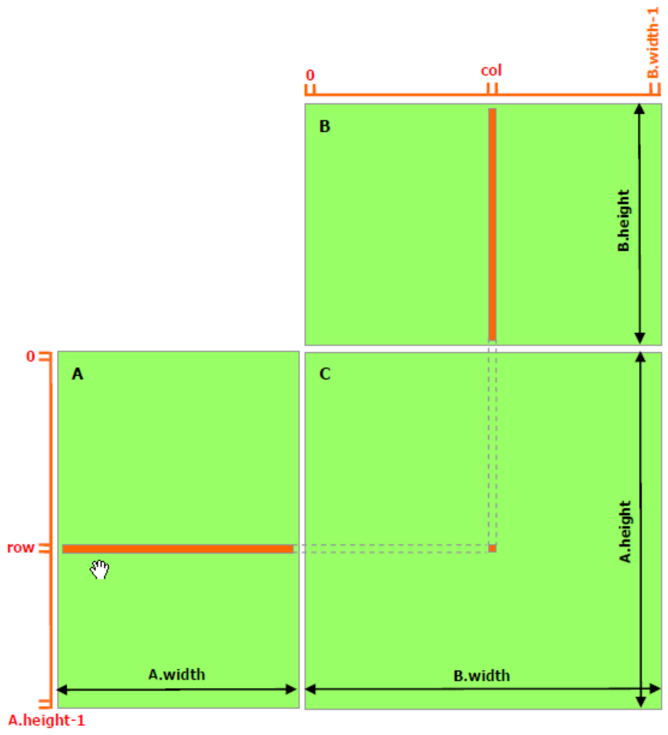


Figure 32: This diagram from the CUDA Guide visualizes a step in matrix multiplication that does not use shared memory.

When calculating element C[i+1][j] for instance the same col [j] from matrix B will have to be accessed again from global memory. Calculating the entire result matrix will therefore result in i and j amount of identical memory accesses to global memory, all being equally slow.

Figure 33 from the CUDA guide illustrates the same calculation, using shared memory, as well as tiling. Here, the calculations are performed block-wise. All data needed to calculate the results in that block are loaded onto shared memory. This includes all rows in matrix A sharing a [j] index with the block indices and all cols in matrix B sharing a [i] index. The elements from matrix A are stored in one shared memory array, and elements from matrix B in another.

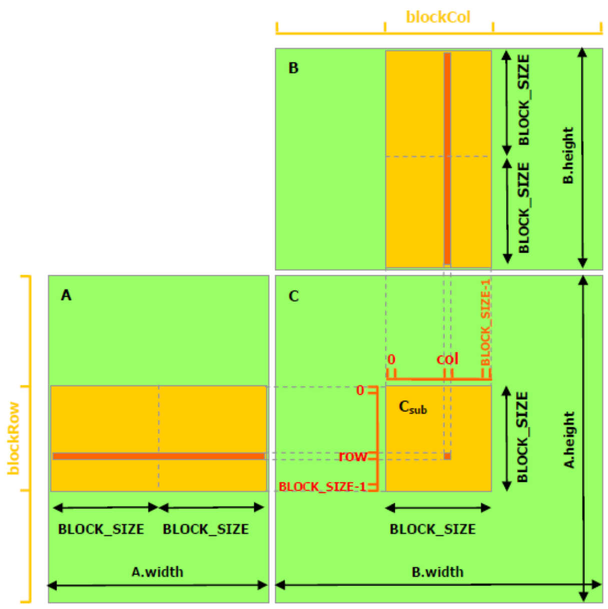


Figure 33: This diagram from the CUDA guide visualizes a step in matrix multiplication that uses shared memory.

This implementation only requires one global memory access per element needed from the input matrices. The rest of the j-1 and i-1 accesses are through shared memory.

For a simple shared memory implementation, the shared memory needed is dependant on the input matrix sizes, but needs to be (A.width × block-size) and (block-size × B.height) (A.width and B.height being the same). With big matrices, this requirement can get very large, and shared memory is limited.

The approach illustrated in figure 33 however, also benefits from tiling, this being achieved by only storing one (block-size × block-size) sized sub-matrix of matrix A and B at the time, and performing all calculations needed with that data, before loading new sub-matrices into shared memory, and performing calculations on those. In the illustrated case, two sub-matrices are used from each input matrix. The result element is saved to matrix C once all sub-matrices have been iterated over, and all calculations have been made.

Using tiling, the shared memory needed will always be of size (block-size × block-size) × 2, and when being mindful of this when determining block-sizes will not cause more shared memory to be requested than is available.

```
__global__ void SharedMemoryAndTiling(float *M1, float *M2, float *M3)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float sharedMemory1[256];
    __shared__ float sharedMemory2[256];
    int sharedIndex = threadIdx.y * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    for (int i = 0; i < M1Cols; i += blockDim.x)
    {
        // Load M1 into shared memory
        if (row < M1Rows && (i + threadIdx.x) < M1Cols)
            sharedMemory1[sharedIndex] =
                M1[row * M1Cols + i + threadIdx.x];
        else
            sharedMemory1[sharedIndex] = 0;

        // Load M2 into shared memory
        if ((i + threadIdx.y) < M1Cols && col < M2Cols)
            sharedMemory2[sharedIndex] =
                M2[(i + threadIdx.y) * M2Cols + col];
        else
            sharedMemory2[sharedIndex] = 0;

        __syncthreads();

        // Tile multiplication
        int numIterations =
            (M1Cols - i > blockDim.x) ? blockDim.x : M1Cols - i;

        for (int j = 0; j < numIterations; j++) {
            sum +=  sharedMemory1[threadIdx.y * blockDim.x + j] *
                    sharedMemory2[j * blockDim.x + threadIdx.x];
        }
        __syncthreads();
    }

    if (row < M1Rows && col < M2Cols) {
        M3[row * M2Cols + col] = sum;
    }
}
```

Figure 34: The CUDA kernel *SharedMemoryAndTiling* for matrix multiplication on a GPU.

The kernel *SharedMemoryAndTiling* on figure 34 benefits from both shared memory and tiling. As the kernel is launched with block-sizes of 16×16, the two shared memory arrays are set to have a size of 256 on lines 5-6, and loaded with data from global memory on lines 12-24. The tiling happens within the loop spanning from line 10 to 37, where tiles are loaded in increments of block-width. All calculations needed for the given tile are performed on lines 29-35.

The last line of each loop iteration calls *__syncthreads()*. This makes sure that all threads executing the kernel are synced up before moving on to the next loop iteration. This might cause some threads to have to wait for others, but is needed, as the all threads within the same block share the shared memory, and all threads must be finished accessing the data before new data is loaded in. The same method is used after loading data into shared memory, to ensure that the data is finished loading before any calculations are performed by the threads. The accumulated *sum* value persist throughout all loop iterations for each thread performing calculations, and is saved to the result matrix on line 40.

The expected performance improvement stems from a combination of the shared memory utilization and the tiling. The shared memory provides much faster memory access, and tiling optimizes this even further by ensuring all elements to be loaded from global memory only once. A cost of using tiling is the overhead caused by the thread synchronization calls, but should be out-weighted by the benefits.

The observed performance of the shared memory kernel showed a significant improvement compared to the parallel kernel on the 1060, while not showing any significant change on the 2080 and 3070. As seen on figure 12 the 1060 has CC 6.1, meaning that the unified data cache for shared memory and L1 is not present, as it is on the other two GPUs.

The introduction of the unified data cache also meant an improved L1 cache, that is able to perform similarly as the shared memory. The GPU might already be storing the frequently accessed elements in the improved L1 cache, and therefore already implementing this speedup in the parallel implementation. This might explain the fast running time of the parallel implementation, and the lack of improvement when utilizing shared memory.

As the 1060 does not have this improved data cache, even with the assumption that the L1 cache is storing all needed elements, there is still possible improvement when fetching elements from stored memory, compared to from the L1.

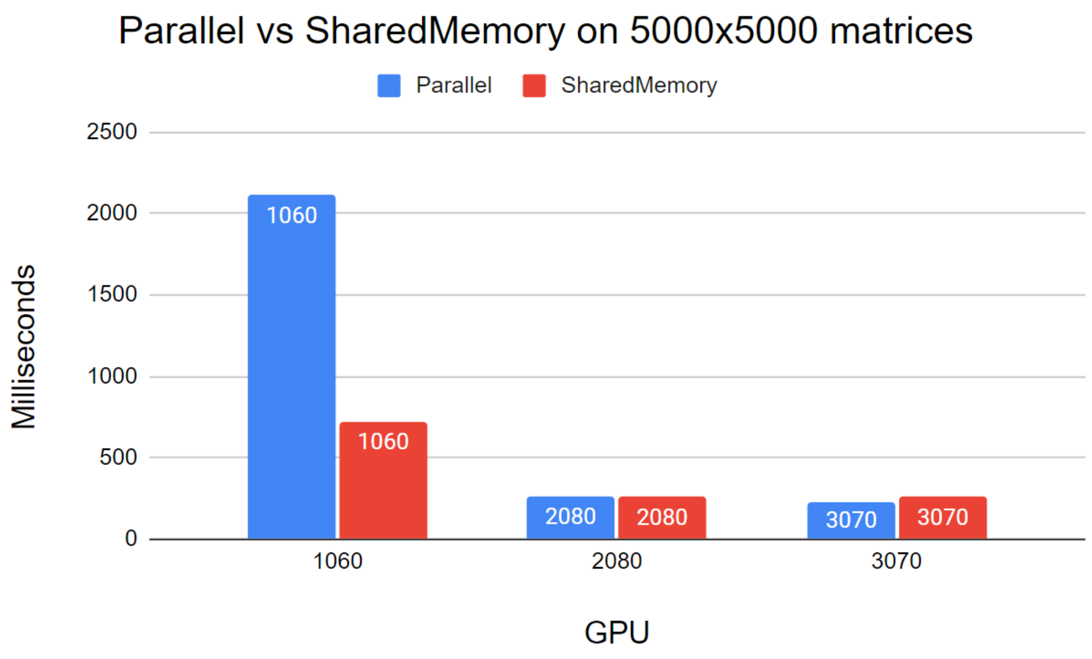


Figure 35: Parallel vs shared memory implementations in CUDA.

4.2.5 CUDA - Investigating further optimization possibilities

As part of our ongoing testing process, we investigated performance using NNC. When analyzing kernel performance for addition, the program suggested increasing the size of the blocks to increase *occupancy*. Occupancy is a metric that measures the number of active warps per SM against the maximum number of possible active warps. It is not a guaranteed way to improve performance, but it can help hide memory latency, by making sure that there is always a warp that is ready to execute an instruction[8].

Inspired by this suggestion, we decided to experiment with different sizes of blocks for multiplication as well. We also experimented with using two-dimensional shared memory. We decided to investigate this based on the hypothesis that it would cause better memory access patterns and fewer bank conflicts. Finally, we combined the two approaches. For simplicity, the optimization attempts are named V1, V2 and V3. We will compare them against the initial version of the *SharedMemory* kernel, which will be dubbed V0.

- **V0**: Original shared memory
- **V1**: Different block sizes
- **V2**: Two-dimensional shared memory
- **V3**: Combining V1 and V2

First, we will go over the three attempts at optimization, then we will compare their performance.

Optimization attempt V1 uses blocks of size 32 × 32 instead of 16 × 16. The CUDA C++ Best Practices Guide[8] discusses Thread and Block Heuristics. It states that there are many factors to consider when determining the optimal block size. An example of such a factor is that higher occupancy does not necessarily lead to better performance, as lower occupancy kernels will have more registers available per thread, which can improve memory accesses. The guide concludes that experimentation will eventually be required, to determine the most optimal configuration parameters.

The V1 kernel uses larger blocks. The kernel launch configuration are given blockDim_32 and gridDim_32, which can be seen in figure 36. The *SharedMemoryAndTiling* kernel can be used again, with the small modification of changing the *sharedMemory* arrays to size 1024, such that they are still able to contain an entire block.

```
dim3 blockDim_32(32, 32);
dim3 gridDim_32((M3Cols + blockDim_32.x - 1) / blockDim_32.x,
                (M3Rows + blockDim_32.y - 1) / blockDim_32.y);
```

Figure 36: Block and grid dimensions for V1.

The V2 kernel uses two-dimensional shared memory instead of one-dimensional, which can be seen in figure 37. The size of each shared memory array is still 256. The memory accesses are updated accordingly, but other than that, the kernel works the same way.

```
__shared__ float sharedMemory1[16][16];
__shared__ float sharedMemory2[16][16];
```

Figure 37: V2 uses two-dimensional shared memory.

V3 combines the strategies used in V1 and V2. It uses two-dimensional shared memory and blocks of size 32 × 32, which can be seen in figure 38.

```
__shared__ float sharedMemory1[32][32];
__shared__ float sharedMemory2[32][32];
```

Figure 38: The V3 kernel uses larger block and grid dimensions as well as correspondingly larger two-dimensional shared memory.

The column chart in figure 39 compares the performance of the kernels when doing multiplication on matrices of size 5000 × 5000. For the 1060 and the 2080, each new iteration improved performance slightly.

Surprisingly, this was not the case for the 3070. Based on our previous results in chapter 4.2.4, we expected V0 to perform worse than the *Parallel* kernel. What we did not expect, was that two-dimensional shared memory (V2) would not only perform better than one-dimensional shared memory (V0), but also the *Parallel* kernel. Interestingly, the 3070 also saw a decrease in performance, when increasing the block size.

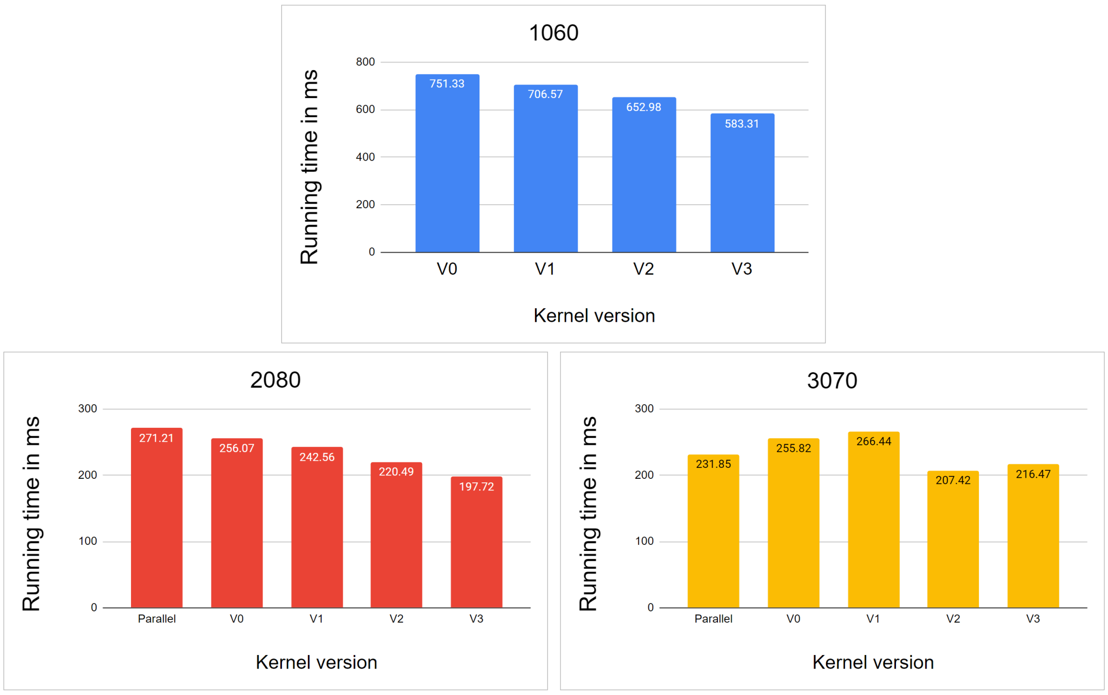


Figure 39: Comparing the performance of the different kernel versions on matrices of size 5000 × 5000.

4.3 Lower-Upper decomposition

4.3.1 A baseline implementation of lower-upper decomposition in C

Our first implementation of lower-upper decomposition (LUD) in C is a straightforward implementation based on the implementation found in *Numerical Recipes in C* [11], although without the Crout's method of pivoting. As the resulting triangular matrices only store non-zero data in non-overlapping element indices (except the unit diagonal of L), we chose to store the results in a single matrix, by overwriting the original matrix A. This solution is memory efficient and follows the advice of minimizing data transfer, from the CUDA Guide. To visualise the L and U matrices, the resulting matrix A can be split such that all elements on the diagonal and above represent the matrix U, where the elements below the diagonal are zeros. Elements below the diagonal represent the matrix L, where elements on the diagonal are 1s and elements above the diagonal are zeros.

```c
 1  void LUD_Simple(float **A, int n) {
 2      for (int i = 1; i <= n; i++) {
 3          for (int j = i; j <= n; j++)
 4          {
 5              float sum = A[i][j];
 6              for (int k = 1; k < i; k++) {
 7                  sum -= A[i][k] * A[k][j];
 8              }
 9              A[i][j] = sum;
10          }
11
12          for (int j = i+1; j <= n; j++)
13          {
14              float sum = A[j][i];
15              for (int k = 1; k < i; k++) {
16                  sum -= A[j][k] * A[k][i];
17              }
18              A[j][i] = sum / A[i][i];
19          }
20      }
21  }
```

Figure 40: The C function *LUD_Simple* for matrix LUD on a CPU.

The implementation in figure 40 consists of multiple loops:

- The first loop (line 2-20) iterates over the matrix A row-wise. This loop represents the main step, where each row is decomposed in turn.

Within the main loop two different loops iterate over the matrix column-wise

- The first loop (line 3-10) begins at the current row i and continues to the last row n. This loop computes the upper triangular matrix U using the formula:

  $$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}$$

  * The k loop (line 6-8) accumulates the sum for the current element $A[i][j]$ by subtracting the contributions from previously computed elements

  The resulting sum is then saved into the $U[i][j]$ element

- The second loop (line 12-19) computes the lower triangular matrix using the formula:

  $$\alpha_{ij} = \frac{1}{\beta_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj} \right)$$

  * The k loop (line 15-17) again accumulates the sum for the current element $A[i][j]$

  The resulting sum is then saved into the $L[i][j]$ element

Both formulas are from *Numerical Recipes in C* [11].

The resulting matrix A will then hold elements of both L and U as seen on figure 41.

$$A = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ l_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ l_{31} & l_{32} & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & u_{nn} \end{bmatrix}$$

Figure 41: The decomposed matrix A. It contains the data for the resulting L and U matrices.

Where:

- The elements u_{ij} (with $i \leq j$) represent the components of the upper triangular matrix U.
- The elements l_{ij} (with $i > j$) represent the components of the lower triangular matrix L, excluding the diagonal of L which is known to be ones.

4.3.2 Adding partial pivoting

As previously mentioned, techniques such as partial pivoting, can be applied to the LUD implementation, to enhance numerical stability. As LUD includes division steps, a problem that could arise include division by zero and division by small numbers. To help avoid this, the rows within the matrix can be swapped during the decomposition.

Even though we did not originally intend to optimize the C implementations of our matrix calculations, we have implemented a second LUD in C, using partial pivoting, to have a more stable version to compare the CUDA implementations with. The pivoting steps added are based on the pivoting implementations in *Afternotes on Numerical Analysis*[2].

```
1  void LUD_Simple_Partial_Pivoting(float** A, int n) {
2      for (int i = 1; i <= n; i++) {
3  
4          //Find pivot row
5          int pivotRow = i;
6          float maxVal = fabs(A[i][i]);
7          for (int p = i + 1; p <= n; p++) {
8              if (fabs(A[p][i]) > maxVal) {
9                  maxVal = fabs(A[p][i]);
10                 pivotRow = p;
11             }
12         }
13 
14         //Swap rows if needed
15         if (pivotRow != i) {
16             for (int j = 1; j <= n; j++) {
17                 float temp = A[i][j];
18                 A[i][j] = A[pivotRow][j];
19                 A[pivotRow][j] = temp;
20             }
21         }
22 
23         //Perform LUD
24         ...
25     }
26 }
```

Figure 42: The C function *LUD_Simple_Partial_Pivoting* for matrix LUD on a CPU.

The implementation seen on figure 42 iterates over the matrix A row-wise, and performs the LUD in the same manner as the simple version (line 23-24). Before the decomposition of each column, a new step is added, beginning on line 5. The current row i is set to being the pivot row, and the diagonal $\mathbf{A}$(i,i) element is set as the current maximal value. When performing the later division calculations, the goal is to divide by as big of a number as possible, so for each row below row i, a check is performed to find the highest absolute value on the same column index

$[j]$. If a higher value is found, the row with the highest value on the $[j]$ column index is swapped with the current row (line 16-20) and the next decomposition step is performed.

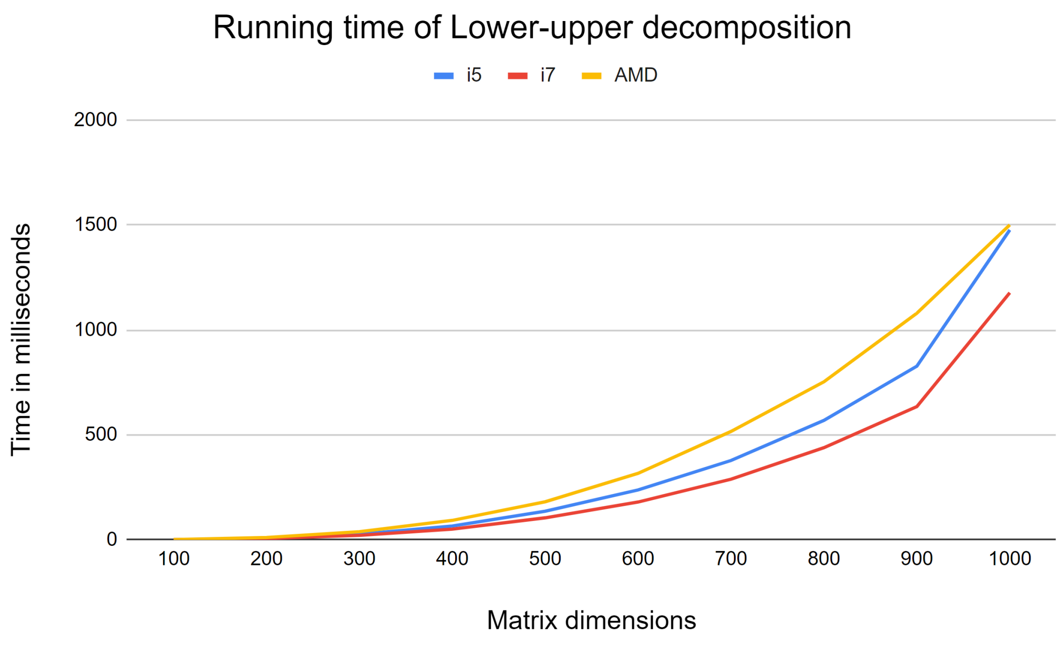


Figure 43: Running time of the *LUD_Simple_Partial_Pivoting* function on the three CPUs.

4.3.3 CUDA - Sequential implementation

The first CUDA implementation follows the same logic as the simple C implementation, with a key difference being that instead of accumulating the result in a *sum* variable, the element is updated directly within each loop iteration. This version of LUD is a right looking LUD algorithm using Gaussian Elimination and is inspired by *Effective GPU Strategies for LU Decomposition* (H. M. D. M. Bandara & D. N. Ranasinghe)[12] and *Afternotes on Numerical Analysis* (G. W. Stewart)[2].

There are multiple factors to take into consideration, when choosing to use this algorithm instead of the algorithm we use in our C implementation. Some of the benefits we considered includes:

Pros

- **Parallelism:** Although there are data dependencies, it allows for concurrent updates of the sub matrix after each column is computed.
- **Fewer synchronization points:** As the threads can operate more independently, the need for thread synchronization is lower.

Other factors to consider when implementing LUD in CUDA include:

- **Numerical stability:** Division can lead to problems when working with small numbers.
- **Floating-point precision:** The amount of operations performed on each element, especially in the lower right portion of the matrix, can accumulate large rounding errors.
- **Memory usage** Compared to matrix multiplication, there are less opportunities to make use of shared memory and caching.

```
__global__ void Sequential(float* A, int n) {
   for (int i = 0; i < n; i++) {

      // Compute L elements (lower triangular part)
      for (int j = i + 1; j < n; j++) {
         A[j * n + i] = A[j * n + i] / A[i * n + i];
      }
      // Compute U elements (upper triangular part)
      for (int j = i + 1; j < n; j++) {
         for (int k = i + 1; k < n; k++) {
            A[j * n + k] = A[j * n + k] - A[i * n + k] * A[j * n + i];
         }
      }
   }
}
```

Figure 44: The CUDA kernel *Sequential* for matrix LUD on a GPU.

For the lower triangular matrix (**L**), elements in the i*th* column below the pivot element (the diagonal $A[i][i]$ element) are calculated by dividing each $A[j][i]$ element by the diagonal element.

Then for the upper triangular matrix (**U**), all elements to the right of and bellow the pivot element are updated, by subtracting the product of the corresponding row element in **U** and the column element in **U**:

$$\mathbf{A[j][k] = A[j][k] - A[i][k] \cdot A[j][i]}$$

This process updates elements for the i*th* row, while row $i + i...n$ is one operation closer to its final result.

These two steps are repeated n times, until all rows and columns of the matrix have been updated, resulting in a combined LU Decomposed matrix, as in the C implementation.

4.3.4 CUDA - Pivoting

As previously mentioned, implementing LUD without any form of stabilisation can result in incorrect or missing results, due to the numerous division operations involved in the computation.

In our CUDA implementation, we used the same partial pivoting method as in the C code. The methodology here is identical: we find the index with the highest absolute value for the elements below the diagonal in each column. If there is an index with a higher absolute value than the *maxVal*, the row containing that value, is swapped with the current row. The only noticeable difference in the code is how the matrix is stored and accessed. In CUDA, matrix **A** is represented as a one-dimensional array on the device, rather than a two-dimensional array on the host, resulting in a slightly different index notation in the code.

```
__global__ void Sequential_With_Partial_Pivoting(float* A, int n) {
    for (int i = 0; i < n; i++) {

        // Find pivot row
        int pivotRow = i;
        float maxValue = fabsf(A[i * n + i]);

        for (int p = i + 1; p < n; p++) {
            if (fabsf(A[p * n + i]) > maxValue) {
                maxValue = fabsf(A[p * n + i]);
                pivotRow = p;
            }
        }

        // Swap rows if needed
        if (pivotRow != i) {
            for (int j = 0; j < n; j++) {
                float temp = A[i * n + j];
                A[i * n + j] = A[pivotRow * n + j];
                A[pivotRow * n + j] = temp;
            }
        }

        __syncthreads();

        // Compute L elements (lower triangular part)
        for (int j = i + 1; j < n; j++) {
            ...
        }
        // Compute U elements (upper triangular part)
        for (int j = i + 1; j < n; j++) {
            ...
        }
    }
}
```

Figure 45: The CUDA kernel *Sequential_With_Partial_Pivoting* for matrix LUD on a GPU.

Matrix / Processor	100	200	300	400	500
i5	1.01	7.99	27.05	66.19	136.70
i7	0.83	6.49	21.46	51.22	105.18
AMD	1.38	11.43	38.99	92.96	181.66
1060	45.77	350.37	1171.59	▨	▨
2080	10.09	103.69	362.25	855.70	1680.84
3070	9.27	85.03	329.70	792.55	1557.82

Figure 46: The running time of the sequential implementations of LUD on all three GPUs and all three CPUs.

Figure 46 shows the running time of the sequential implementations that use partial pivoting. Just as seen with matrix addition and matrix multiplication, the sequential approach on the GPUs has significantly worse performance than the CPUs.

4.3.5 CUDA - Parallelization

The first step in making a parallel version of the LUD implementation, was identifying what steps we are able to parallelize[8], and separate those from the ones that are not.

Looking at the sequential implementation in figure 44, the outer i loop needs to be iterated through sequentially. This is because each iteration is dependent upon the results from the previous one. The two j loops we are able to parallelize, as each element updated within those loops are not dependent upon each other. The second $l$oop responsible for the **U** elements is however dependent upon the updated **L** elements from the first j loop, and they can therefore not be run concurrently.

The pivoting in figure 45 could also benefit from parallelization, when swapping rows on line 16-22, as entire rows could be swapped concurrently. Finding the pivot row is still done sequentially, although this could also be parallelized as well.

To parallelize these 3 steps, they are separated into 3 different kernels. The three kernels are shown in figure 48, 51 and 52.

In attempting to optimize LUD using a GPU, it is important to consider Amdahl's law, which states that the speedup of parallelizing a process is always limited by its sequential parts[13]. This means that the outer i loop will always be our primary constraint. Even if we achieve significant speedup by parallelizing sections of the code, the speedup is bounded by the time spent performing this sequential loop.

Therefore to achieve the best possible performance, the rest of the LUD process takes place on host, in the *Parallel_Pivoted* function seen in figure 54. By doing this, we are able to to utilize the sequential capabilities of the CPU and the parallel capabilities of the GPU, thereby limiting the constraints of the sequential loop, by maximizing the strengths of each processing unit. We are also able to launch each kernel with the appropriate amount of threads and block/grid dimensions, and we can change those parameters for each launch. Doing this also follows the heterogeneous programming model, which the CUDA guide explains to be an assumed part of the CUDA programming model. (An illustrated example of heterogeneous programming can be seen in appendix 10.1)

Before starting the LUD, all matrix data is copied from host to device, and once finished, the data is copied back from device to host. This means that while performing the LUD, the data cannot be accessed or computed from the host, without an additional copy back and forth. As this copying step would negatively impact the performance, all data computation is performed on the device, while the host mainly functions as a launcher for the appropriate kernels. Because of this, the step of finding the pivot row has to be performed on the device, despite it not being a parallel version, and this step is therefore also added as a kernel (figure 47).

```
__global__ void FindPivot(float* A, int* pivotIndices, int n, int i) {
    int pivotRow = i;
    float maxValue = abs(A[i * n + i]);

    for (int p = i + 1; p < n; ++p) {
        float value = abs(A[p * n + i]);
        if (value > maxValue) {
            pivotRow = p;
            maxValue = value;
        }
    }
    pivotIndices[i] = pivotRow;
}
```

Figure 47: The CUDA kernel *FindPivot* for finding the pivot row on a GPU.

```
__global__ void SwapRows(float* A, int* pivotIndices, int n, int i) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    for (int col = tid; col < n; col += blockDim.x * gridDim.x) {
        float temp = A[i * n + col];
        A[i * n + col] = A[pivotIndices[i] * n + col];
        A[pivotIndices[i] * n + col] = temp;
    }
}
```

Figure 48: The CUDA kernel *SwapRows* for swapping rows on a GPU.

The *FindPivot* kernel is launched with a single thread, as we previously discovered that multiple threads performing the same sequential task negatively impacts performance. Once the pivot has been found, its row number is stored in the *pivotIndices* array, which holds the pivoting order for the entire LUD process. By storing these indices, we are able to reconstruct the original matrix.

The *SwapRows* kernel is then launched with a number of threads equal to the number of elements per row in the matrix **A**. These threads are separated into row-like blocks of dimensions $(blockDim.x, 1)$, and in a grid of dimension $((n + blockDim.x - 1)/blockDim.x, 1)$, with *blockDim.x* being 32. Each thread is then responsible for swapping one element in the ith row with the corresponding element from the pivot row. The thread's placement in the grid structure indicates what element in the row it responsible for swapping.

Versions of the Pivot & Swap method

Our first version of pivoting on the GPU was done in a single kernel that handled both finding the pivot and swapping the rows. This kernel was then launched with a single thread, performing the entire pivot and swap process in a sequential manner. As this process happened in each iteration of the outermost i loop, this was not an optimal solution, because it slowed down the entire process significantly. By Amdahl's law, this meant our potential speedup of the entire LUD process became constrained by this step.

For the next version, we switched to a more parallel version, launching the still combined kernel with the row-like blocks in a grid large enough to have threads of equal or larger amount than elements in each row of matrix **A**. The pivot element would still be found sequentially by a single thread (specified to thread with the index $(0, 0)$), and then the swapping of elements was performed in parallel by all the threads. This version resulted in a significant speedup, but it became apparent that it would be better to split the two tasks into two different kernels. This avoids $n - 1$ threads idling, while one thread performs the first task.

In the final version with the two kernels, the first kernel is launched with a single thread like the first version. The second kernel is launched with the custom grid size from the second version. This resulted in another significant performance improvement. The performance of all three versions can be seen in figure 49, where the entire LUD was run on square matrices of different sizes, using the different versions.

For even further optimization, a future version of the *findPivot* kernel could be parallelized as well. This could be done using a parallel reduction algorithm. The kernel would launch multiple threads to perform concurrent comparisons of element pairs, and select the highest absolute value in each pair. The amount of active threads would then be reduced to half, to perform a new comparison of the highest elements from the previous comparisons. This process would be repeated until only one element remains. As this approach would require multiple access to the same elements in each comparison iteration, shared memory would be utilized to store the elements and their indices. An example of how this approach could look can be seen in figure 50.

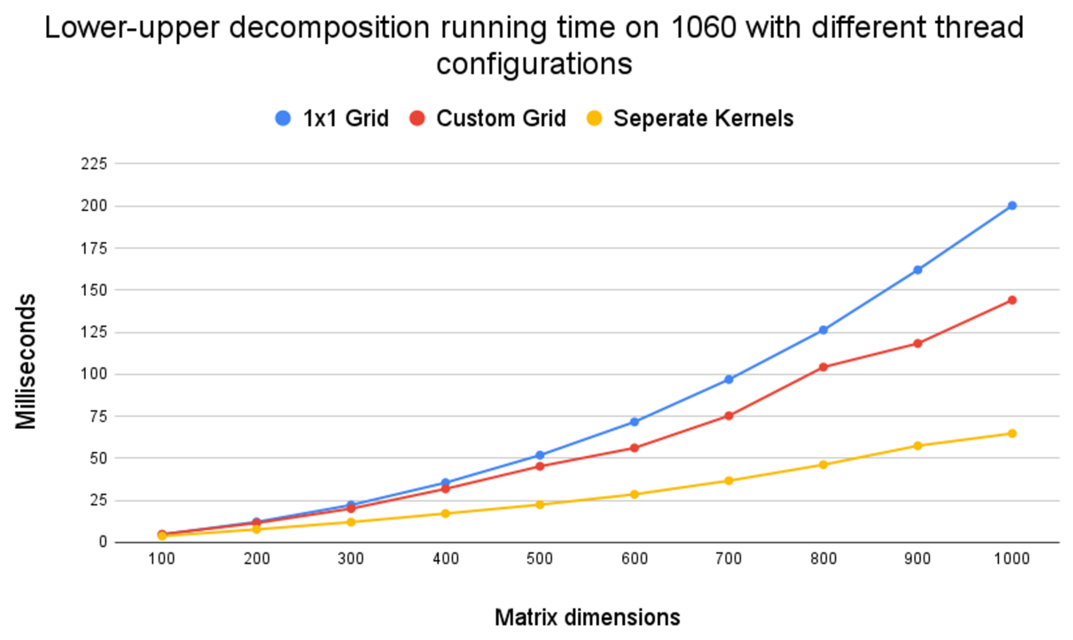


Figure 49: The running time of LUD with different versions of Pivot & Swap implementations.

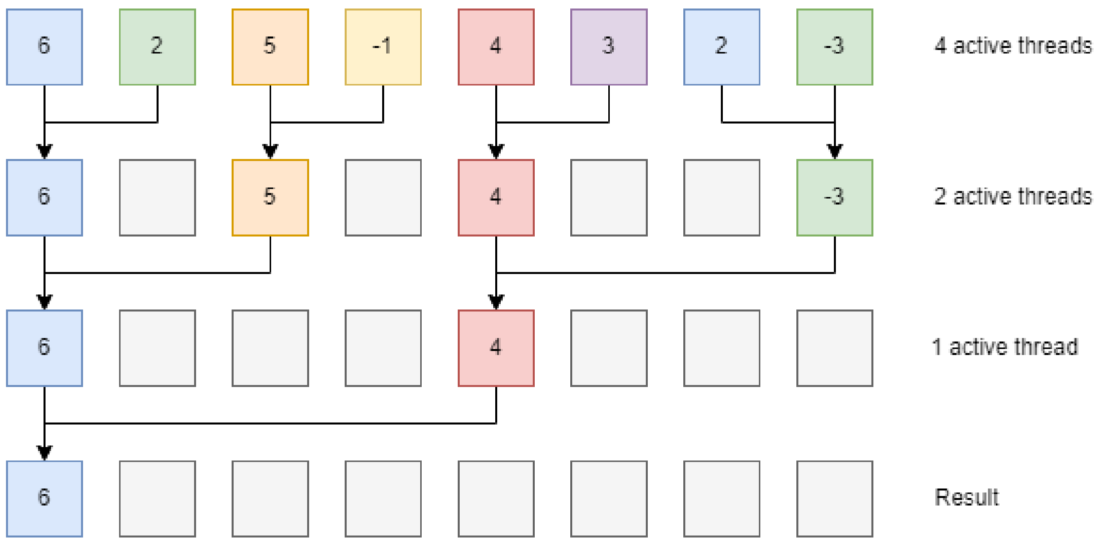


Figure 50: An example of a parallel reduction algorithm for finding the pivot element.

The *ComputeLowerColumn* kernel is launched with column-like blocks of dimension $(1, blockDim.y)$ in a grid with dimensions $(1, (subMatrixDim + blockDim.x - 1)/blockDim.x)$, $blockDim.x$ and $blockDim.y$ being 32. Just as in the swapping kernel, each thread is responsible for calculating the corresponding index.

```
__global__ void ComputeLowerColumn(float* A, int n, int i) {
    int row = blockIdx.y * blockDim.y + threadIdx.y + i + 1;

    if (row < n) {
        A[row * n + i] /= A[i * n + i];
    }
}
```

Figure 51: The CUDA kernel *ComputeLowerColumn* for computing column i on a GPU.

The *UpdateSubmatrix* kernel is launched with more standard block dimensions of (32,32), and a grid that is large enough to cover the submatrix that is being updated.

```
__global__ void UpdateSubmatrix(float* A, int n, int i) {
    int row = blockIdx.y * blockDim.y + threadIdx.y + i + 1;
    int col = blockIdx.x * blockDim.x + threadIdx.x + i + 1;

    if (row < n && col < n) {
        A[row * n + col] -= A[i * n + col] * A[row * n + i];
    }
}
```

Figure 52: The CUDA kernel *UpdateSubmatrix* for updating the submatrix for the next ith iteration on a GPU.

Figure 53 illustrates the kernels and grid dimensions used for an ith iteration. The red rectangle represents a diagonal element $\mathbf{A}(i, i)$. The blue blocks represent the grid used for launching the *ComputeLowerColumn* kernel, and the yellow blocks represent the grid used for launching the *UpdateSubmatrix* kernel.

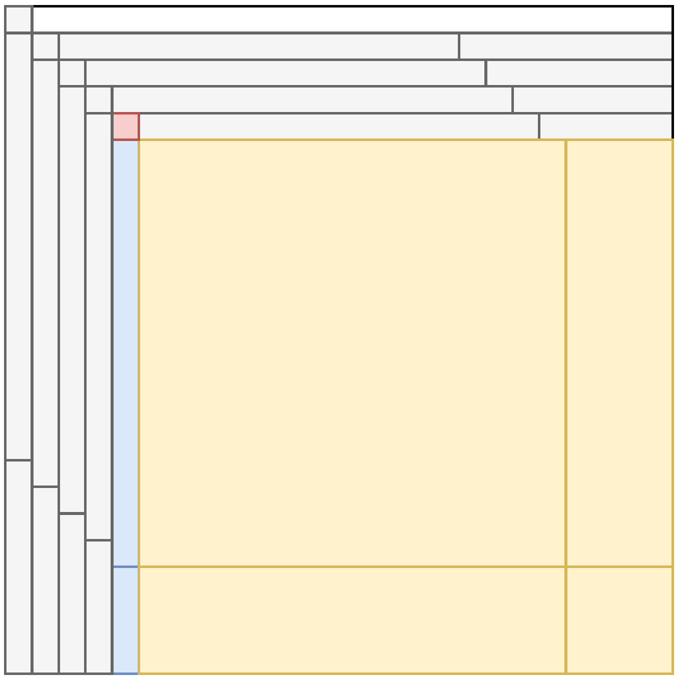


Figure 53: Grid dimensions used for LUD kernels.

The host CPU function *Parallel_Pivoted* can finally be seen in figure 54. The host starts by creating the output array of pivot indices and allocates space for this on the device (line 3 & 4), and then performs the LUD by launching the kernels with the previously described grids. Lastly, it returns the array of pivot indices.

```
int* Parallel_Pivoted(float* A, int n, dim3 blockDim) {

    int* pivotIndices;
    cudaMalloc(&pivotIndices, n * sizeof(int));

    // Loop over each row - Must be done 1 at the time
    for (int i = 0; i < n; i++) {

        // Find pivot and swap
        dim3 blockDimRow(blockDim.x, 1);
        dim3 gridDimRow((n + blockDim.x - 1) / blockDim.x, 1);
        FindPivot <<<1,1>>> (A, pivotIndices, n, i);
        SwapRows <<<gridDimRow, blockDimRow >>> (A, pivotIndices, n, i);
        cudaDeviceSynchronize();

        //Dimensions of the submatrix below/to the right of element (i,i)
        int subMatrixDim = n - i - 1;

        //Setup grid and launch ComputeLowerColumn
        dim3 blockDimColumn(1, blockDim.y);
        dim3 gridDimColumn(1, (subMatrixDim + blockDim.x - 1)
            / blockDim.x);
        ComputeLowerColumn <<<gridDimColumn, blockDimColumn>>> (A, n, i);

        //Setup grid and launch UpdateSubmatrix
        dim3 gridDimSubmatrix((subMatrixDim + blockDim.x - 1)
            / blockDim.x, (subMatrixDim + blockDim.y - 1) / blockDim.y);
        UpdateSubmatrix <<<gridDimSubmatrix, blockDim>>> (A, n, i);
    }

    int* hostPivotIndices = (int*)malloc(n * sizeof(int));
    cudaMemcpy(
        hostPivotIndices, pivotIndices,
        n * sizeof(int), cudaMemcpyDeviceToHost
    );

    cudaFree(pivotIndices);
    return hostPivotIndices;
}
```

Figure 54: The host CUDA function *Parallel_Pivoted* for a CPU. It calls multiple kernels as part of its execution.

A note on numerical stability

We implemented partial pivoting to improve the numerical stability of the LUD, as the number of operations, especially divisions, could lead to inaccurate results due to accumulative rounding errors. But even with the partial pivoting, we still experienced that on larger matrices, rounding errors did become apparent, especially in the right bottom parts of the resulting matrix.

For our testing, matrices are populated with:

```
matrix.data[i * matrix.cols + j] = (float) rand() / rand() + 0.00001f;
```

On a matrix of size $1000\times$ 1000, the accumulative error across all elements is approximately 41727, where 3469 elements (0.3469%) have a difference higher than 1.0 from the input matrix. This was calculated by multiplying the output **L** and **U** matrix and comparing each element of the result with the elements from the original matrix. We might be able to reduce the errors by implementing full pivoting, but when using 32-bit floating points, there will always be some errors due to the finite precision of floats. One way to combat such errors is switching to 64-bit. The reason not to do so, is the difference in processing power for single precision and double precision on GPUs, as was seen in figure 12.

4.3.6 CUDA - Shared memory

Finding ways to utilize shared memory started with identifying where the same memory accesses were happening multiple times. This is not the case for the pivoting process, as each element is only accessed once during the swapping, and the process is happening across multiple blocks.

The kernels that could possibly benefit from shared memory are the *ComputeLowerColumn* and *UpdateSubmatrix* kernels. These can be seen in the version in figure 55.

```
__global__ void ComputeLowerColumnShared(float* A, int n, int i) {
    __shared__ float pivotElement;
    pivotElement = A[i * n + i];

    int row = blockIdx.y * blockDim.y + threadIdx.y + i + 1;

    if (row < n) {
        A[row * n + i] /= pivotElement;
    }
}

__global__ void UpdateSubmatrixShared(float* A, int n, int i) {
    __shared__ float sharedRow[32];
    __shared__ float sharedCol[32];

    int row = blockIdx.y * blockDim.y + threadIdx.y + i + 1;
    int col = blockIdx.x * blockDim.x + threadIdx.x + i + 1;

    if (row < n && threadIdx.x == 0) {
        sharedCol[threadIdx.y] = A[row * n + i];
    }
    if (col < n && threadIdx.y == 0) {
        sharedRow[threadIdx.x] = A[i * n + col];
    }

    __syncthreads();

    if (row < n && col < n) {
        A[row * n + col] -=
            sharedRow[threadIdx.x] * sharedCol[threadIdx.y];
    }
}
```

Figure 55: The CUDA kernels *ComputeLowerColumnShared* and *UpdateSubmatrixShared* for a GPU. They have been updated to utilize shared memory.

In the *ComputeLowerColumnShared* kernel, a single float is stored in shared memory, this being the diagonal element $\mathbf{A}(i, i)$. Each thread divides an element in the column below the diagonal by this shared element. The memory access to the shared element element is reduced from one global memory access per thread, here being $n - i - 1$ accesses, to only one access per block.

In the *UpdateSubmatrixShared* kernel, each block is responsible for computing a specific part of the submatrix. Each thread accesses an element in the first row of the block, and in the column to the left of the submatrix. Because of this, this row and column are loaded into shared memory, reducing the memory access from one global memory access per thread for both the row and the column, to just one per block. It is however also important to note the overhead introduced, especially when synchronizing threads, as seen on line 26 in figure 55.

When considering the many steps involved with the LUD process, the amount of shared memory use here is not nearly as significant as in our matrix multiplication and addition implementations, and therefore has less impact on the performance.

The shared memory implementation is then run from the host function in the same manner as the parallel implementation (figure 54, swapping the *ComputeLowerColumn* kernel with the *ComputeLowerColumnShared* kernel, and the *UpdateSubmatrix* kernel with the *UpdateSubmatrixShared* kernel.

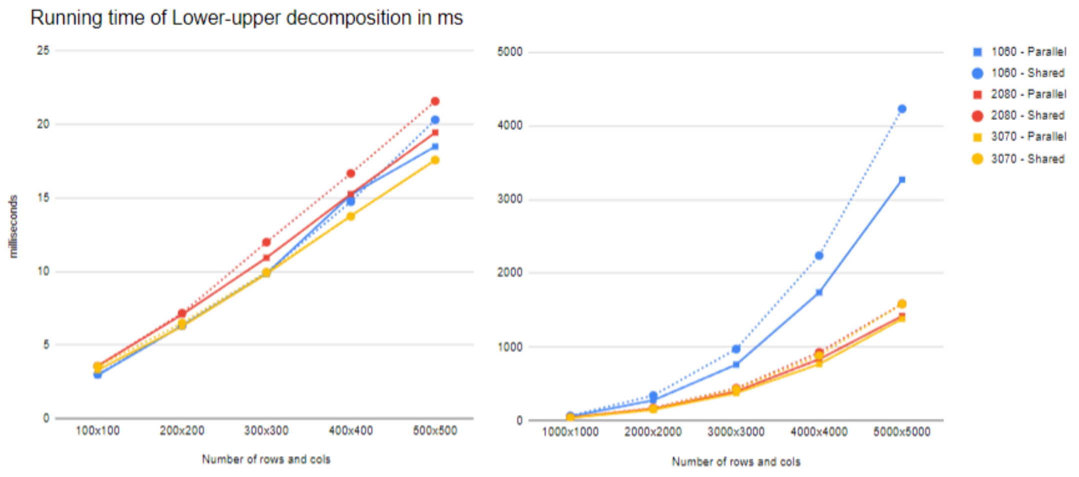


Figure 56: Running time of LUD on the three GPUs.

As can be seen in figure 56, using shared memory did not improve performance. While it being comparatively equal in performance on smaller matrices, it became slower than the parallel implementation on larger matrices, especially on the 1060. This could be a result of the memory access patterns when loading data into shared memory being inefficient, or bank conflicts when threads are accessing elements in shared memory. Even if the usage of shared memory might optimize some memory accesses, our implementation seems to have caused more overhead than speedup, resulting in a negative performance impact. It is possible that exploring different memory access patterns would give an performance improvement, but this would have to be tested further. For our final comparison of the CPU and GPU performance, we will be using the performance data from the parallel implementation without shared memory usage.

Matrix / Processor	100	200	300	400	500	1000
i5	1.01	7.99	27.05	66.19	136.70	1476.90
i7	0.83	6.49	21.46	51.22	105.18	1177.51
AMD	1.38	11.43	38.99	92.96	181.66	1500.98
1060	2.97	6.33	9.86	15.21	18.49	59.65
2080	3.60	7.08	10.94	15.26	19.43	45.74
3070	3.31	6.27	9.84	13.78	17.57	42.17

Figure 57: Running time of the sequential CPU implementation vs the parallel GPU implementation.

As can be seen in figure 57, the CPU implementation is faster on smaller matrices, but the performance scales much better with the GPU implementation. When comparing the fastest CPU with the slowest GPU we see speedup of factor approximately 20 on a matrix of size 1000x1000 and a speedup of factor 35 on system 3 (AMD and 3070). It is of course important to keep in mind that following a heterogeneous programming model. some parts of the CUDA implementation is performed on the CPU. The data shows that utilizing the GPU to perform the large tasks is significantly faster, and the combined utilization of the systems is very beneficial, especially on large matrices.

5 Performance

5.1 GPU performance gains versus transfer time cost

To determine if GPU utilization was a net improvement, we need to include the cost of transferring data to and from the GPU. An important factor to keep in mind, is that we have not tried to optimize CPU performance. It is possible that better CPU implementations could outperform one or all kernels for a given task. To get a definitive answer, one would have to test the best possible CPU implementation against the best possible GPU implementation (including transfer time). Since that is outside the scope of this project, we will investigate how our kernels stack up against an initial CPU implementation, which requires much less work to set up.

Figure 58 shows the transfer time we achieved when transferring two square matrices to the GPU and transferring a result matrix back. This time also includes the necessary setup, i.e. allocating memory on the GPU and setting the grid and block dimensions.

For addition, the transfer time turned out to have a considerable impact on performance. Addition took around 317 milliseconds on the i7 and AMD CPUs for square matrices of size 10,000, where parallel addition on the GPU took around 3 ms. While using the GPU is still faster overall, around 150 ms of transfer time is 50 times higher than the actual GPU computation time.

Looking at multiplication and LUD, it is obvious that transfer time becomes an afterthought, as arithmetic intensity increases. For multiplication, using the GPU is the obvious choice. The CPUs were already spending 700 to 900 ms on square matrices of size 600 x 600, where the GPUs took just 0.5 to 2.5 ms. When doing Lower-Upper Decomposition, the transfer time is even lower than the tests we performed, as there is just one input matrix instead of two. Again, when the input matrix is 1000 x 1000, the CPUs take around 1000-2000 ms, where the GPUs take around 50 ms, the transfer time is not a concern.

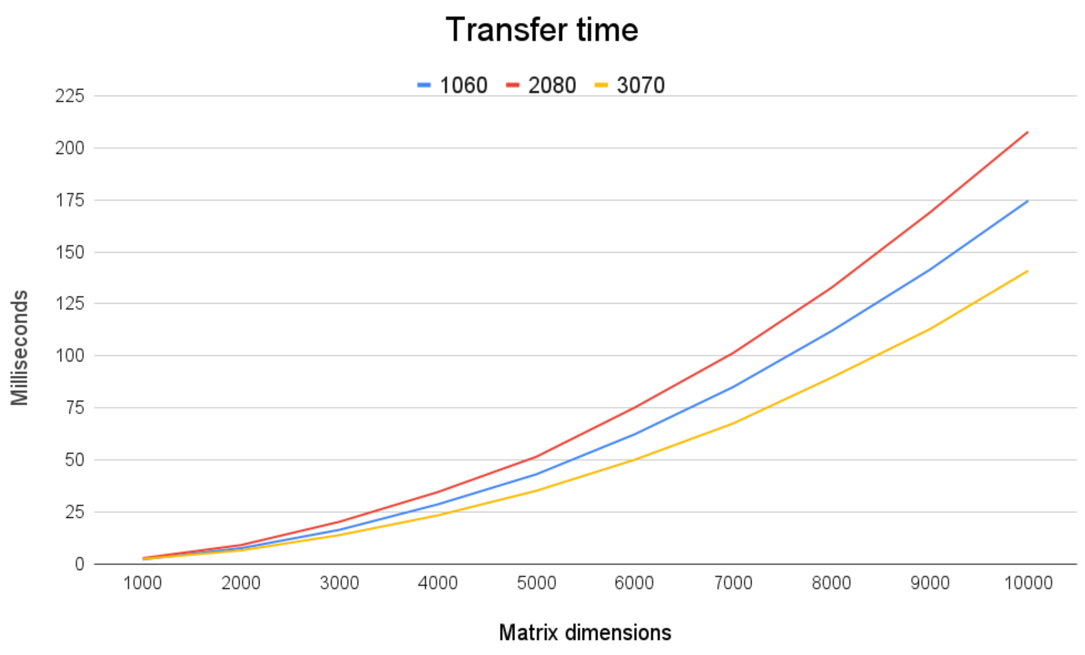


Figure 58: The achieved transfer time for two square matrices. The time includes transfer of two input matrices to the GPU and transfer of one result matrix back to the CPU.

5.2 Comparing achieved and theoretical performance

In this chapter, we will compare the performances of the implementations against the theoretical FLOPS performance of the GPUs. The FLOPS executed by the kernels have been gathered using NNC[3]. We had to exclude the 1060 as it is not supported.

To analyze the TFLOPS it is crucial to understand the *fused multiply-add* (FMA) instruction. As the name suggests, this is an instruction that, enabled by specific hardware, can perform multiplication and addition in a single cycle. The hardware also performs add and multiplication instructions separately. This means that the amount of instructions that are multiply-add, will have a significant impact on performance. To reach 100% of theoretical performance, it is necessary to exclusively use multiply-add [14]. NNC provides data about overall use of the FMA hardware, but does not specify how much of it is multiply-add.

We will now analyze select kernels to get insight on metrics that are relevant to the performance and go into further details. For addition we will inspect the sequential kernel. For multiplication we will look at the parallel and shared memory (V3) kernels.

[3]Figure 67 in the appendix shows an example output for *Floating Point Operations Roofline* in NNC.

5.2.1 Addition

The tables in figure 59 compares the achieved and theoretical performance of the GPUs. The achieved performance for addition peaks at only 0.364 % of the theoretical performance, which is disappointingly low.

Addition - 2080

Implementation	Throughput (TFLOPS)	% of theoretical (10.13)
Sequential	0.0000189	0.000186 %
Parallel	0.037	0.364 %
Shared Memory	0.037	0.364 %

Addition - 3070

Implementation	Throughput (TFLOPS)	% of theoretical (11.36)
Sequential	0.0000111	0.0000977 %
Parallel	0.036	0.317 %
Shared Memory	0.035	0.308 %

Figure 59: The percentage of theoretical performance achieved by the addition kernels on the 2080 and the 3070. The Sequential kernels are executed by a single thread.

The sequential performance being much better when utilizing a single thread, suggests that the FMA computation is not a limiting factor, as more available threads would offer more FMA units. To investigate this, we ran the kernel with multiple threads. Inspecting the row chart from 'Compute Workload Analysis' in figure 60 when using multiple threads, shows the Pipe Utilization as a percentage of active cycles and peak instructions executed. The *% of active cycles* shows how often a type of pipeline was utilized during active cycles, while the *% of peak instructions executed* shows how close to theoretical performance a given pipeline executed. From the chart, it is evident that the FMA pipeline is not anywhere near full utilization. It was only used 9.65 % of active cycles.

Meanwhile, the LSU pipeline achieved 89.05 % of peak instructions executed, meaning that it ran close to its theoretical capabilities. LSU stands for Load Store Unit. The LSU handles fetching and saving data to memory[15]. The high utilization is an indication that the kernel is heavily memory-bound.

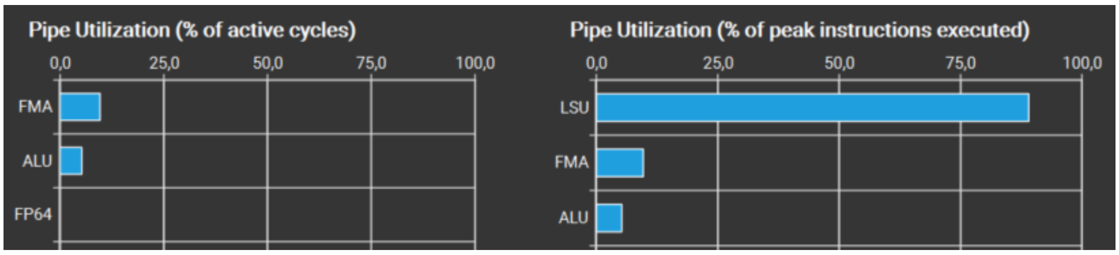


Figure 60: A snippet of the 'Compute Workload Analysis' when running the addition kernel with multiple threads.

To further cement this hypothesis, we inspected the *Floating Point Operations Roofline*, which can be seen in figure 61. The *single precision roofline* (circled in white) is the theoretical maximum performance. The green dot (circled in red) is the *single precision achieved value*. If the achieved value is to the right of the single precision roofline, the kernel is compute-bound[4]. If it is to the left of the roofline, as in this case, it is memory-bound[16].

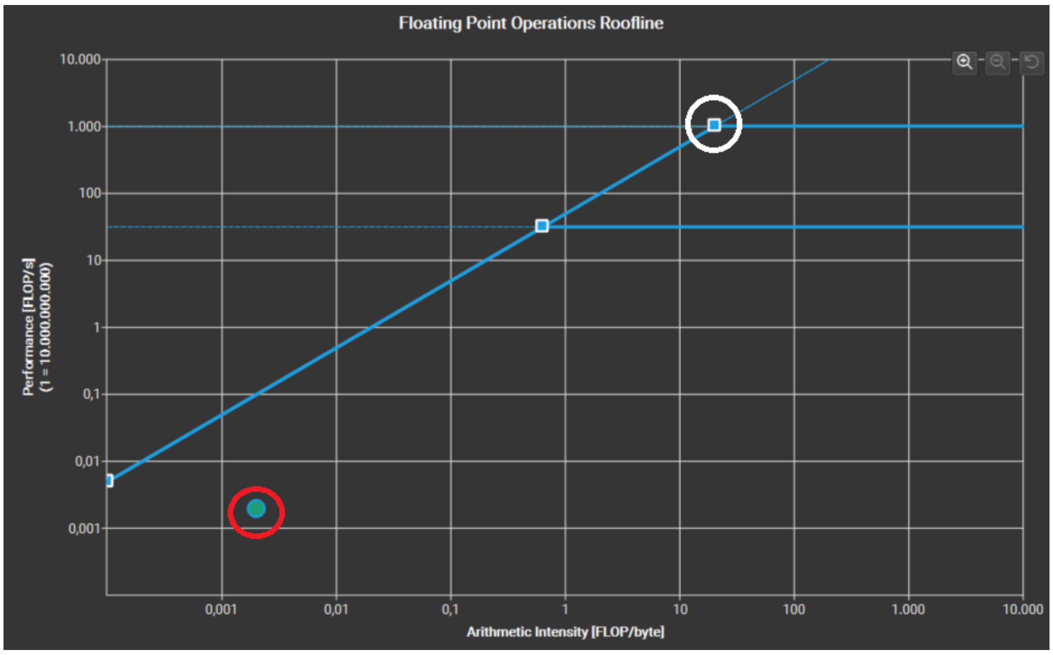


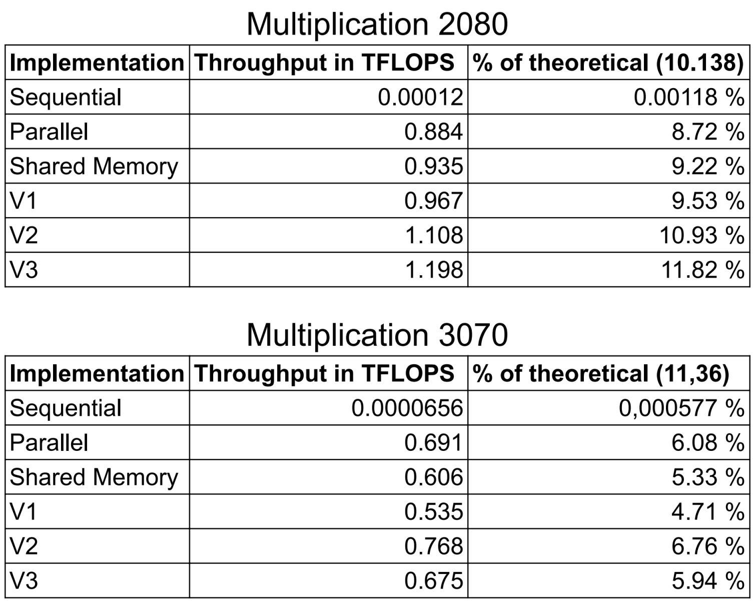
Figure 61: The *Floating Point Operations Roofline* for addition.

We also inspected sequential performance when running the kernel with a single thread. This time, the LSU performed 6.88 % of peak instructions executed, while the FMA hit 0.72 %. Again, the LSU performed closer to its maximum capability than the FMA did.

[4]Figure 68 (from the *Kernel Profiling Guide*) in the appendix, visually demonstrates the *Memory Bound Region* and the *Compute Bound Region*.

5.2.2 Multiplication

For the multiplication results in figure 62, the peak achieved performance i 11.82 % of theoretical performance, which is pleasing.



Multiplication 2080

Implementation	Throughput in TFLOPS	% of theoretical (10.138)
Sequential	0.00012	0.00118 %
Parallel	0.884	8.72 %
Shared Memory	0.935	9.22 %
V1	0.967	9.53 %
V2	1.108	10.93 %
V3	1.198	11.82 %

Multiplication 3070

Implementation	Throughput in TFLOPS	% of theoretical (11,36)
Sequential	0.0000656	0,000577 %
Parallel	0.691	6.08 %
Shared Memory	0.606	5.33 %
V1	0.535	4.71 %
V2	0.768	6.76 %
V3	0.675	5.94 %

Figure 62: The percentage of theoretical performance achieved by the addition kernels on the 2080 and the 3070. The sequential kernels are executed by a single thread.

The *GPU Throughput* for shared memory (V3) in figure 63, shows that both the SMs and the memory is working at 75% of theoretical performance. The LSU executed at 75% *of peak instructions executed* again, while FMA reached 15.10%.

More experimentation could potentially improve the amount of FMA operations that are fused multiply-add. When compiling with `nvcc`, the flag `fmad` can be used to turn on or turn off fused multiply-add[17]. By running with fused multiply-add on and off, we could contrast running time. Assuming the flag does not change anything else, this could give more insight into FMA than NNC does.

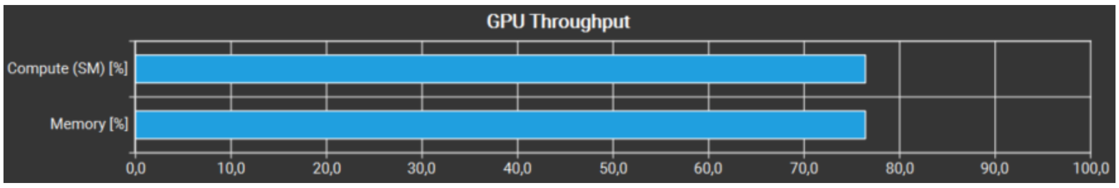


Figure 63: The *GPU Throughput* chart gives an overview of how much the available computational and memory bandwidth was utilized.

For each category that NNC analyzes, it estimates a % speedup, if any potential performance issues are discovered. For the shared memory V3 kernel, it estimated the following speedups. We will discuss the two major ones, *Issue Slot Utilization* and *MIO Throttle* (memory input/output).

Speedup Factor	*Estimated Speedup (%)*
Issue Slot Utilization	72.93
MIO Throttle	51.86
L2 Load Access Pattern	3.26

The Issue Slot Utilization describes how many cycles are used to issue instructions to warps[16]. The instructions to warps are issued by *scheduler*s. A scheduler maintains a pool of warps and is able to issue one or more instructions to one warp each cycle. Every cycle, the scheduler checks the state of the warps in the pool (*Active Warps*). If there are no *Eligible Warps* among the Active Warps, the *issue slot* is skipped and thus, the opportunity to issue an instruction is missed. For our kernel, we achieved an average of an instruction every 3.7 cycles, which means that a lot of issue slots are not utilized.

Our low Issue Slot Utilization is connected to MIO Throttle. Data about MIO Throttle is collected in the *Warp State Statistics* in NNC. MIO Throttle is a metric of how many cycles a warp spends waiting for the MIO instruction queue to not be full[16]. On average, each warp of our kernel is stalled for 15.3 cycles, waiting for the instruction queue to clear up. The average amount of cycles for two instructions to be issued on a warp is 29.5 cycles. This means that 51.9% of cycles are not utilized.

In essence, Issue Slot Utilization is low because there are no Eligible Warps and there are no Eligible Warps, because there is MIO Throttle. This knowledge also offers a reasonable explanation as to why more threads causes addition to run significantly slower.

The Kernel Profiling Guide from NVIDIA suggests using fewer but wider loads when accessing shared memory, to combat MIO Throttle. If we had more time to optimize the kernel, this would be a good place to start. We will not discuss the L2 Load Access Pattern as it is only a fraction of the potential speedup. Hypothetically, if we could implement optimize the kernel and achieve the estimated speedups, we could reach upwards of 3.224 TFLOPS.

$$1.189 \times (1 + \frac{3.26}{100}) \times (1 + \frac{51.86}{100}) \times (1 + \frac{72.93}{100}) = 3.224 \text{ TFLOPS}$$

While we do not expect to be able to perform at theoretical performance, it seems plausible that there are still unexplored options to improve achieved performance. We will mention some of them in section 5.3.

To our bewilderment, the 3070 performed around half as well as the 2080 when testing with NNC, despite performing similarly to the 2080 in our own testing environment. At this point, we have been unable to figure out why this happens.

Another potential problem, is that the values of *Compute (SM) [%]* and *Memory [%]* in figure 63 were identical for the multiplication kernels. It seems unlikely that this is a coincidence. Whether this can be attributed to a legitimate bottleneck issue or some kind of testing error is also unclear.

This goes to show that NNC is a very powerful tool, but also very information-dense. There is certainly much more to be explored.

5.2.3 LUD

Analyzing LUD was more difficult than addition and multiplication. Previously, NNC would provide metrics for the entire kernel, but since LUD consisted of multiple kernels, each run was diagnosed independently. Where before there was 1 kernel for a run, there were now 400 kernels for a run. This also meant that overall data for TFLOPS, arithmetic intensity and GPU Throughput were not available.

The limited access to good data, makes it difficult to compare the achieved performance to the theoretical performance. Still, there are some valuable insights.

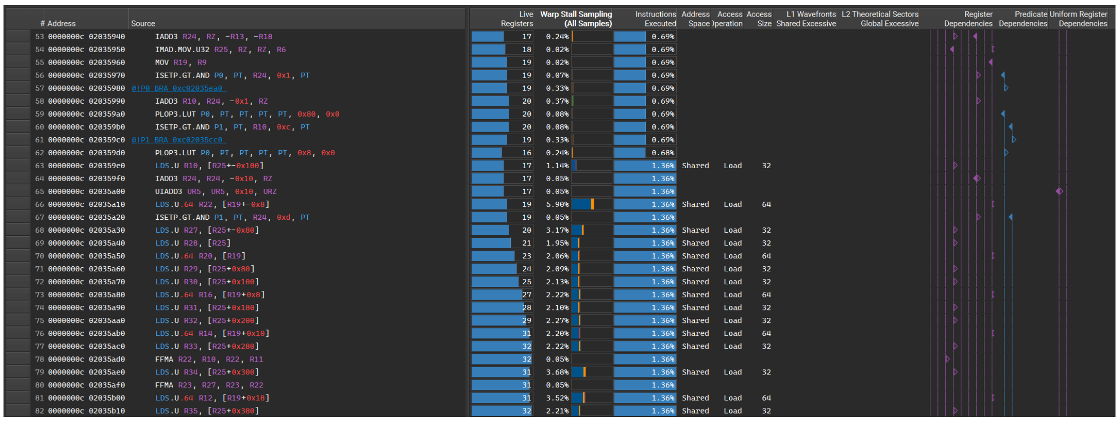
NNC suggests that the small grids do not utilize the full resources of the 48 SMs. While it is technically true, the amount of columns is so low, compared to the amount of columns that would be needed, in order for this optimization to have any measurable impact. NNC is not well suited to diagnose subtasks that only need a fraction of the GPUs power. This is not so much a critique of the design of NNC, but more a lesson that the focus of optimization should be on large computational loads.

It is possible that there are solutions that make analysis more convenient. We did not manage to find them in the project period. One thing we did try, was using implementing LUD a single kernel instead of a function calling multiple kernels. This caused many headaches, as the launch configurations cannot be modified after a kernel launch. This meant that using thread IDs became an extremely difficult task and the idea was subsequently dropped.

5.3 Further hypotheses for reaching higher performance

Since a lot of performance bottleneck comes from limited data transfer bandwidth on the device, it would be interesting to implement more sophisticated solutions for this. One possibility is the *CUDA async copy* feature, which is available on the 3070. This feature allows data to be transferred directly from the L1 cache to shared memory. Moving the data directly from the L1 cache to shared memory removes the time it takes to move the data through registers on the way [18].

Another thing to look into is more advanced warp stall analysis, which NNC also offers. There are multiple potential reason for warp stall, MIO throttle among them, which is described in the Kernel Profiling Guide[16]. Figure 64 shows the output for warp stall analysis. Understanding this tool could help analyze precisely when and how warp stalling occurs.



#	Address	Source	Live Registers	Warp Stall Sampling (All Samples)	Instructions Executed	Address Space	Access)peration	Access Size	L1 Wavefronts Shared Excessive	L2 Theoretical Sectors Global Excessive	Register Dependencies	Predicate Dependencies	Uniform Register Dependencies
53	0000000c 02035940	IADD3 R24, RZ, -R13, -R18	17	0.24%	0.69%								
54	0000000c 02035950	IMAD.MOV.U32 R25, RZ, RZ, R6	18	0.02%	0.69%								
55	0000000c 02035960	MOV R19, R9	19	0.02%	0.69%								
56	0000000c 02035970	ISETP.GT.AND P0, PT, R24, 0x1, PT	19	0.07%	0.69%								
57	0000000c 02035980	@!P0 BRA 0xc02035ea0	19	0.33%	0.69%								
58	0000000c 02035990	IADD3 R10, R24, -0x1, RZ	20	0.37%	0.69%								
59	0000000c 020359a0	PLOP3.LUT P0, PT, PT, PT, PT, 0x80, 0x0	20	0.08%	0.69%								
60	0000000c 020359b0	ISETP.GT.AND P1, PT, R10, 0xc, PT	20	0.08%	0.69%								
61	0000000c 020359c0	@!P1 BRA 0xc02035cc0	19	0.33%	0.69%								
62	0000000c 020359d0	PLOP3.LUT P0, PT, PT, PT, PT, 0x8, 0x0	16	0.24%	0.68%								
63	0000000c 020359e0	LDS.U R10, [R25+-0x100]	17	1.14%	1.36%	Shared	Load	32					
64	0000000c 020359f0	IADD3 R24, R24, -0x10, RZ	17	0.05%	1.36%								
65	0000000c 02035a00	UIADD3 UR5, UR5, 0x10, URZ	17	0.05%	1.36%								
66	0000000c 02035a10	LDS.U.64 R22, [R19+-0x8]	19	5.90%	1.36%	Shared	Load	64					
67	0000000c 02035a20	ISETP.GT.AND P1, PT, R24, 0xd, PT	19	0.05%	1.36%								
68	0000000c 02035a30	LDS.U R27, [R25+-0x80]	20	3.17%	1.36%	Shared	Load	32					
69	0000000c 02035a40	LDS.U R28, [R25]	21	1.95%	1.36%	Shared	Load	32					
70	0000000c 02035a50	LDS.U.64 R20, [R19]	23	2.06%	1.36%	Shared	Load	64					
71	0000000c 02035a60	LDS.U R29, [R25+0x80]	24	2.09%	1.36%	Shared	Load	32					
72	0000000c 02035a70	LDS.U R30, [R25+0x100]	25	2.13%	1.36%	Shared	Load	32					
73	0000000c 02035a80	LDS.U.64 R16, [R19+0x8]	27	2.22%	1.36%	Shared	Load	64					
74	0000000c 02035a90	LDS.U R31, [R25+0x180]	28	2.10%	1.36%	Shared	Load	32					
75	0000000c 02035aa0	LDS.U R32, [R25+0x200]	29	2.27%	1.36%	Shared	Load	32					
76	0000000c 02035ab0	LDS.U.64 R14, [R19+0x10]	31	2.20%	1.36%	Shared	Load	64					
77	0000000c 02035ac0	LDS.U R33, [R25+0x280]	32	2.22%	1.36%	Shared	Load	32					
78	0000000c 02035ad0	FFMA R22, R10, R22, R11	32	0.05%	1.36%								
79	0000000c 02035ae0	LDS.U R34, [R25+0x300]	31	3.68%	1.36%	Shared	Load	32					
80	0000000c 02035af0	FFMA R23, R27, R23, R22	31	0.05%	1.36%								
81	0000000c 02035b00	LDS.U.64 R12, [R19+0x18]	31	3.52%	1.36%	Shared	Load	64					
82	0000000c 02035b10	LDS.U R35, [R25+0x380]	32	2.21%	1.36%	Shared	Load	32					

Figure 64: An example output of warp stalling analysis, which could be used for further investigation into performance.

6 Results

Based on the results of our project, we come to the conclusion that it is possible to increase performance of matrix calculations, when using CUDA on an NVIDIA GPU instead of a CPU. The results have varied for different calculations. On the low end, we achieved a performance increase from about 325 ms to 150 ms execution time for addition. This result is for square matrices of size 10,000 x 10,000, including the time it takes to transfer the data. We also achieved massive performance improvements from around 700 ms down to just 0.54 ms for multiplication. This is for matrices of size 600 and is a performance improvement of about a factor 1300. For LUD, we saw a performance increase from around 1200ms down to 42ms, with a significantly better scaling of execution time for the GPUs, as data size increased.

Based on our experience with this project, we have come to a number of practical recommendations. Most central to the use of GPUs, is that they should be used for workloads that can be executed in parallel. We want to focus on parallel algorithms, where steps in the workload are not effected by previous steps. On the other hand, we want to avoid sequential algorithms, where steps involved in completing the workload depends upon previous steps to be completed first.

We want the size of the workload to be sufficiently large, such that the time investment and added complexity of GPU use is warranted. For example, for smaller workloads like addition or LUD on square matrices of size 100x100, a sequential CPU solution performs about three times faster than a parallel GPU solution. This is not the case for matrices of size 1000x1000, where we achieved around a factor 25 performance improvement with a parallel GPU solution. Additionally, as the ratio between the workload and the data size increases, GPUs become a more attractive strategy, as the cost of data transfer becomes insignificant.

As developers, we often do not have the luxury of thoroughly exploring every possible solution. An example of that is this project itself, where we had to forego optimizations on the CPU, which would have matched the best case CPU performance against the best case GPU performance. In general, we want to use our best judgement to determine where to spend our available time. While NNC is available to help analyze possible optimizations, it is important to keep in mind that the suggestions made by the program is centered around CUDA architecture. Suggestions by the program do not consider if a given solution is well suited for the CUDA architecture or if they are there out of necessity. Some cases will offer great performance increases, where others will be significantly more tricky to implement correctly and may not result in worthwhile performance increases.

When we have chosen where to apply our efforts, it is also worth keeping in mind that there is a steep difficulty curve, once the initial parallel implementation works. A basic parallel implementation on the CUDA architecture is only a couple of lines more code than a sequential CPU implementation. The added complexity of distributing work to threads is small, when compared to the technical knowledge needed to see further performance gains, like analyzing warp stalls and distributing work optimally to the streaming multiprocessors.

7 Discussion

On the whole, the project went as we expected. We developed an understanding for the CUDA architecture, implemented kernels based on the understanding and concluded that there is a good reason for it to exist. It would have been surprising if we found no benefits of using GPUs, since they are already widely used for parallel tasks, including matrix calculations. One of the reasons for the wide use of GPUs, is that they excel at computationally demanding tasks, like running large language models and high resolution video games. With that in mind, the project was more so an exploration of the work to go from a sequential CPU solution to an optimized GPU solution.

While the project went as expected on a macro level, there were lots of things we learned throughout the project, that changed its initial trajectory. Most importantly, we thought that there was much less access to analytical data about kernel performance. Throughout the project, we discovered and learnt more about the performance metrics offered by NVIDIA Nsight Compute. This meant that the analysis became less black box than we anticipated.

In the first half of the project, we oriented our work around testing the performance ourselves and focusing narrowly on optimizing the resulting running time. In the last half, we increasingly broadened the scope of our analysis. While this meant that we did not get to explore the implications of the analysis in section 5, we got to develop a more nuanced understanding of how the kernels performed and what kept them from realizing the full potential of the GPUs. If we were to do the project over, knowing what we do now, it would be interesting to focus exclusively on optimizing individual metrics.

One thing we did not anticipate, was the difficulty of implementing LUD. Developing an understanding of LUD and getting correct outputs with acceptable error margins was a long and frustrating process. This meant that we had to adjust the number of matrix calculations we would explore. Initially, we planned to also implement and test singular value decomposition, but we had to stick to addition, multiplication and LUD to have enough time to do thorough implementation, testing and analysis.

On a micro level, we made errors and missed opportunities along the way. There is a flag we could have used to find minimum, maximum and average metrics of a kernel. Having that information, would have allowed us to identify performance trends when switching to shared memory when analyzing LUD in section 5.2. We also had to cut more corners than we preferred, in getting LUD to work, as we would have liked to achieve better numerical stability.

Finally, we could have saved time testing the impact of block and grid sizes, if we had noticed earlier that these metrics are included in a full detail analysis in NVIDIA Nsight Compute. We did not notice it earlier, as we were running partial analysis. We made the decision to do that, because the time to run the analysis was too long for practical testing. Had we known earlier, we could have run one full metric, which would include the data, that we were manually gathering.

8 Future work

The most obvious way to continue the work we have done during the project period is to implement and test singular value decomposition. This could potentially offer more insight into the CUDA architecture.

Having discovered how much there is to learn about the CUDA architecture, it seems likely that the best use of time would be doing more research and testing. Specifically, trying to optimize the use of shared memory, such that it would provide better performance than a parallel implementation on newer GPUs and CC versions. The biggest leap in performance between parallel and shared memory that we managed to achieve was for the 1060, due to its lack of a unified data cache.

An additional area of interest related to shared memory is bank conflicts. When making the switch to shared memory, it is still important to be mindful of coalesced memory accesses. Working to reduce the amount of bank conflicts would likely lead to a better understanding of the flow of memory in the CUDA architecture and subsequently lead to better performance when utilizing shared memory.

There are also features of newer CC versions that we did not get to explore, like the async-copy feature from CUDA 11.0, that was briefly mentioned.

Furthermore, the hypotheses for reaching higher performance that were mentioned would also be a good next step. Developing an understanding of warp stall analysis could give a foundation for much more sophisticated decision making, because it would likely offer much more detailed analysis of performance bottlenecks.

For specific kernels, it would be beneficial to find a better solution for the LUD implementation, such that we could analyze the overall performance of all kernels for an entire run, as we for now have only been able to gather data for each individual kernel in each individual LUD run. This would make data analysis much easier and likely more detailed. We would also be able to use Amdahl's law to estimate the maximum achievable speedups, based on the distribution of time spent on each step of the computation.

The *findPivot* kernel could be updated to a parallel version using multiple threads and shared memory, to compare all elements of a column in a more efficient way. In its current sequential state, it limits the potential performance increase.

Lastly, the implementation of shared memory for LUD can be tested further, as we know from NNC analysis, as well as the results from H. M. D. M. Bandara & D. N. Ranasinghe [12] that it should be possible to achieve increased performance when using shared memory. Our first steps would include testing different memory access patterns, both when loading elements into shared memory as well as accessing them.

9 Litterature

Sources

[1] GeeksForGeeks. *Doolittle Algorithm : LU Decomposition*. https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/. Accessed: 12-12-2023. 2022.

[2] Gilbert W. Stewart. *Afternotes on Numerical Analysis*. 3rd edition. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1996. ISBN: 0-89871-362-5.

[3] Intel. *CPU vs. GPU: Making the Most of Both*. https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html. Accessed: 16-09-2023.

[4] Nvidia. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 06-06-2023. 2023.

[5] Brian Caulfield. *What's the Difference Between a CPU and a GPU?* https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/. Accessed: 16-09-2023. 2009.

[6] Nvidia. *CUDA Toolkit Archive*. https://developer.nvidia.com/cuda-toolkit-archive. Accessed: 26-10-2023. 2023.

[7] Nvidia. *Your GPU Compute Capability*. https://developer.nvidia.com/cuda-gpus. Accessed: 26-10-2023. 2023.

[8] Nvidia. *CUDA C++ Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#getting-started. Accessed: 17-10-2023. 2023.

[9] Nvidia. *cuBLAS*. https://docs.nvidia.com/cuda/cublas/. Accessed: 24-10-2023. 2023.

[10] Peter Sestoft. *Microbenchmarks in Java and C#*. Accessed: 06-06-2023. 2015.

[11] William H. Press, Brian P. Flannery, Saul A. Teukolsky & William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. 2nd edition. New York, NY, USA: Cambridge University Press, 1992. ISBN: 0-521-43108-5.

[12] H. M. D. M. Bandara & D. N. Ranasinghe. *Effective GPU Strategies for LU Decomposition*. https://www.hipc.org/hipc2011/studsym-papers/1569512927.pdf. Accessed: 25-11-2023. 2011.

[13] Geeks for Geeks. *Computer Organization — Amdahl's law and its proof*. https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/. Accessed: 13-12-2023. 2023.

[14] Nvidia. *Maximizing FLOPS*. https://forums.developer.nvidia.com/t/maximizing-flops/142425. Accessed: 25-11-2023. 2020.

[15] Nvidia. *Pipe Utilization*. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/pipeutilization.htm. Accessed: 26-11-2023. 2015.

[16] Nvidia. *Kernel Profiling Guide*. https://docs.nvidia.com/nsight-compute/2020.1/pdf/ProfilingGuide.pdf. Accessed: 26-11-2023. 2020.

[17] Nvidia. *NVIDIA CUDA Compiler Driver NVCC*. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-cuda-compilation. Accessed: 28-11-2023. 2023.

[18] Matthieu Tardy & Carter Edwards. *Controlling Data Movement to Boost Performance on the NVIDIA Ampere Architecture*. https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/. Accessed: 13-12-2023. 2020.

10 Appendix

10.1 Heterogeneous Programming

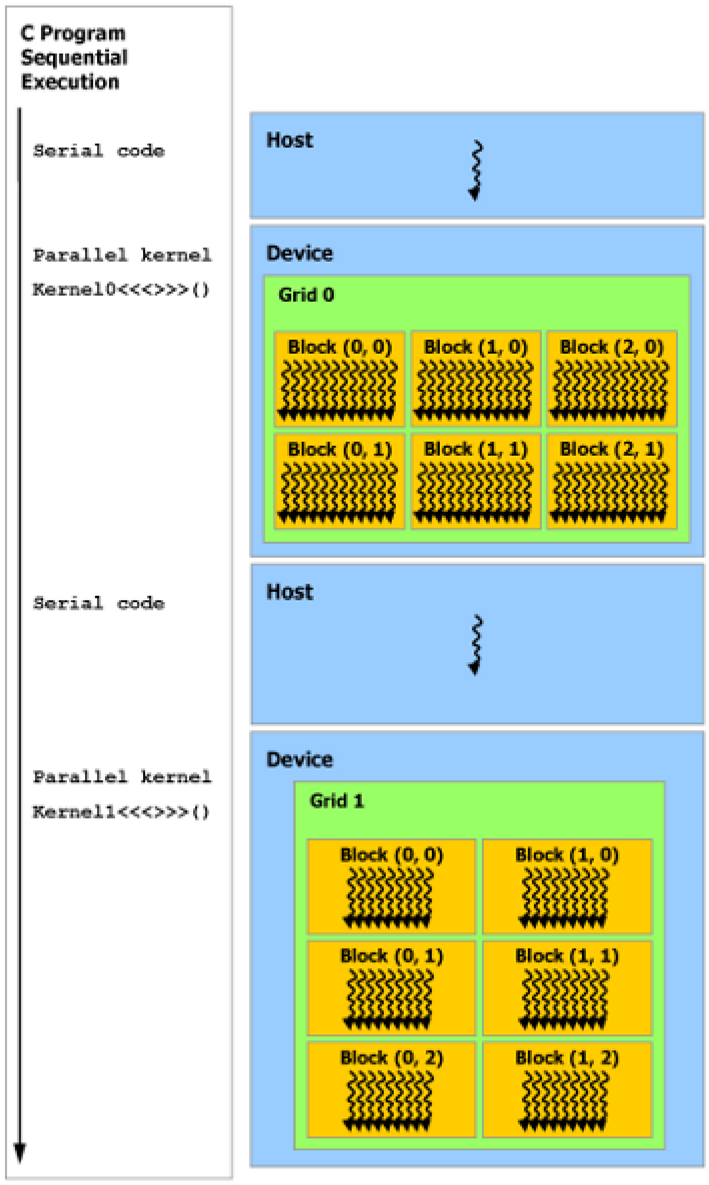


Figure 65: An illustration of the CUDA programming model from the CUDA guide, highlighting the execution of sequential code on the host CPU and parallel kernels on the device GPU, with separate memory spaces and thread organization into grids and blocks[4].

10.2 CUDA measurement implementation

The following code is how we measure the performance of CUDA kernels. Multiple kernels can be tested with calls to *measureExecutionTimes*, which can be seen on line 32-34.

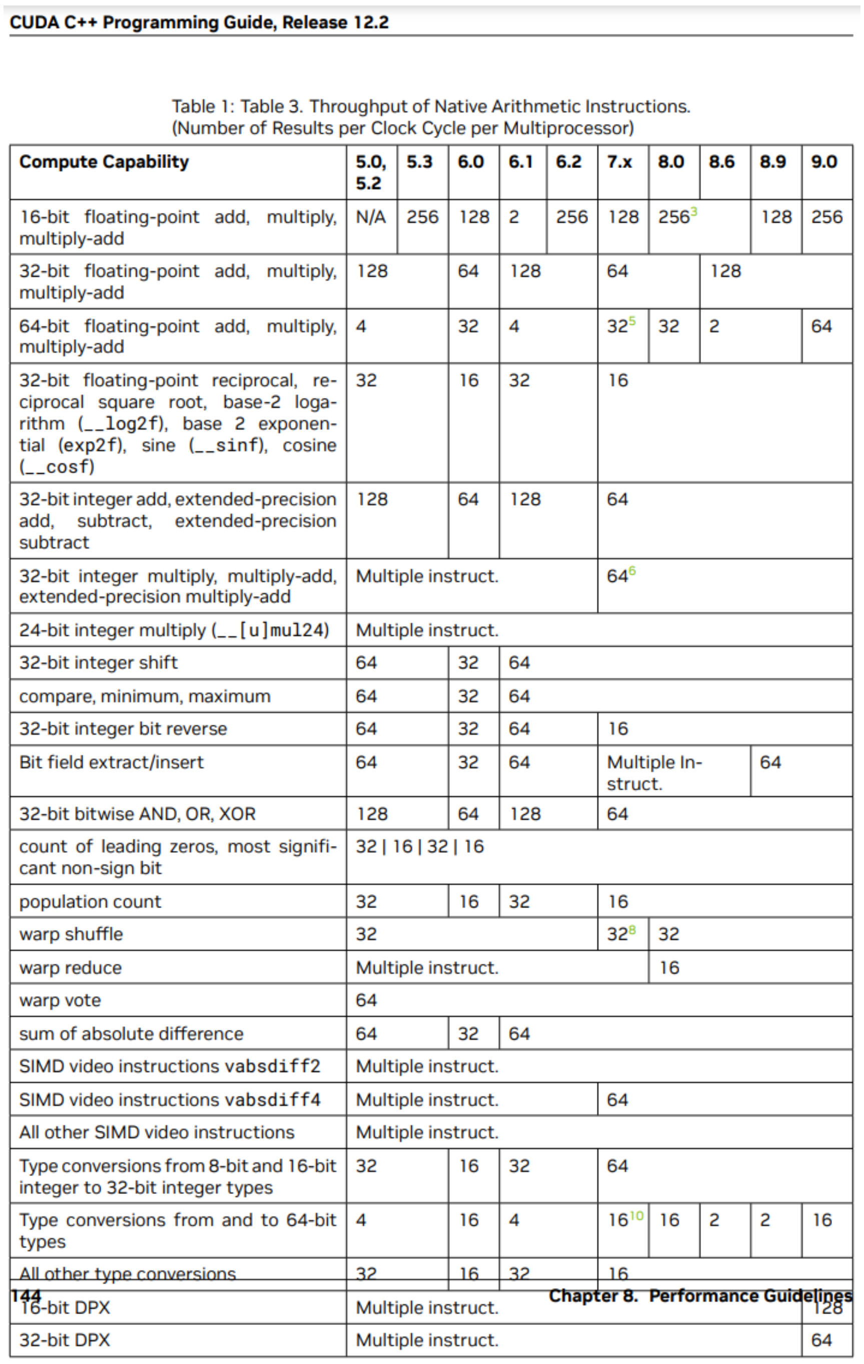
```
1  int main(int argc, char* argv[])
2  {
3      int MRows = atoi(argv[1]);
4      int MCols = atoi(argv[2]);
5      size_t memorySize = MRows * MCols * sizeof(float);
6  
7      // Timer measure time spent on a process
8      Timer timer = createTimer();
9  
10     beginTimer(timer);
11     Matrix M1, M2, M3;
12     float *device_M1, *device_M2, *device_M3;
13     initializeMatricesAndMemory(M1, M2, M3, MRows, MCols, MRows, MCols,
           ↪ MRows, MCols);
14     allocateMemoryOnGPU(device_M1, device_M2, device_M3, memorySize,
           ↪ memorySize, memorySize);
15     copyMatricesToGPU(M1, M2, device_M1, device_M2, memorySize,
           ↪ memorySize);
16     endTimer(timer, "initialize matrices on CPU and GPU",
           ↪ printDebugMessages);
17 
18     // Define block and grid dimensions for CUDA kernel
19     dim3 blockDim(16, 16);
20 
21     if (MRows <= 16 && MCols <= 16)
22     {
23         blockDim = dim3(MCols, MRows); // Use matrix size for smaller
               ↪ matrices
24     }
25 
26     dim3 gridDim((MCols + blockDim.x - 1) / blockDim.x, (MRows + blockDim
           ↪ .y - 1) / blockDim.y);
27 
28     // Create an array to store execution times for each kernel
29     float executionTimes[3][100]; // 3 kernels, 100 executions each
30 
31     // Measure and record execution times for all kernels
32     measureExecutionTimes(executionTimes[0], Sequential,     device_M1,
           ↪ device_M2, device_M3, MRows, MCols, 1, 1);
33     measureExecutionTimes(executionTimes[1], Parallel,       device_M1,
        ↪ device_M2, device_M3, MRows, MCols, gridDim, blockDim);
    measureExecutionTimes(executionTimes[2], SharedMemory,  device_M1,
        ↪ device_M2, device_M3, MRows, MCols, gridDim, blockDim);

    // Copy the result matrix from device to host
    cudaMemcpy(M3.data, device_M3, memorySize, cudaMemcpyDeviceToHost);

    // Open a new file to write the result into
    char fileName[100];     // Max length of filename

    // Customize filename to reflect size of result matrix
    sprintf(fileName, "Test/Floats_Execution_Times_Matrix_Size_%dx%d.csv"
        ↪ , MRows, MCols);
    FILE *outputFile = fopen(fileName, "w");
    if (outputFile == NULL)
    {
        perror("Unable to create the output file");
        return 1;
    }

    // Write execution times to the output file in separate columns
    fprintf(outputFile, "Sequential,Parallel,SharedMemory\n");
    for (int i = 0; i < 100; i++)
    {
        fprintf(outputFile, "%f,%f,%f\n",
                executionTimes[0][i],
                executionTimes[1][i],
                executionTimes[2][i]);
    }

    // Close the output file
    fclose(outputFile);

    freeMemory(device_M1, device_M2, device_M3, M1, M2, M3);

    // Exit program
    return 0;
}
```

10.3 C measurement implementation

The following code is an example of how we measure the performance of C functions.

```c
int main(int argc, char *argv[])
{
    int MRows = atoi(argv[1]);
    int MCols = atoi(argv[2]);
    size_t memorySize = MRows * MCols * sizeof(float);

    // Start measuring time OS spends on process
    C_Timer timer = create_C_Timer();

    // Initialize matrices
    Matrix M1 = create_C_Matrix(MRows, MCols);
    Matrix M2 = create_C_Matrix(MRows, MCols);
    Matrix M3 = create_C_Matrix(MRows, MCols);

    // Read data into M1 and M2
    populateWithRandomFloats(M1);
    populateWithRandomFloats(M2);

    // Array to store execution times for 100 iterations
    double executionTimes[100];

    for (int i = 0; i < 100; i++)
    {
        beginTimer(&timer);
        // Perform addition
        additionSequential(M1, M2, M3);
        // Add measured time for this iteration
        executionTimes[i] = endTimer(&timer);
    }

    // Open a new file to write result into
    char filename[100];
    snprintf(filename, 100, "Test/Addition_Floats_Runtime_Matrix_Size_%dx
        ↪ %d.csv", MRows, MCols);

    FILE *outputFile = fopen(filename, "w");
    if (outputFile == NULL)
    {
        perror("Unable to create the output file");
        return 1;
    }
    else
    {
        for (int i = 0; i < 100; i++)
        {
            fprintf(outputFile, "%f\n", executionTimes[i]);
        }
    }

    // Deallocate memory
    free(M1.data);
    free(M2.data);
    free(M3.data);

    // Close result file and end program
    fclose(outputFile);
    return 0;
}
```

10.4 Throughput of Native Arithmetic Instructions for different Compute Capabilities



CUDA C++ Programming Guide, Release 12.2

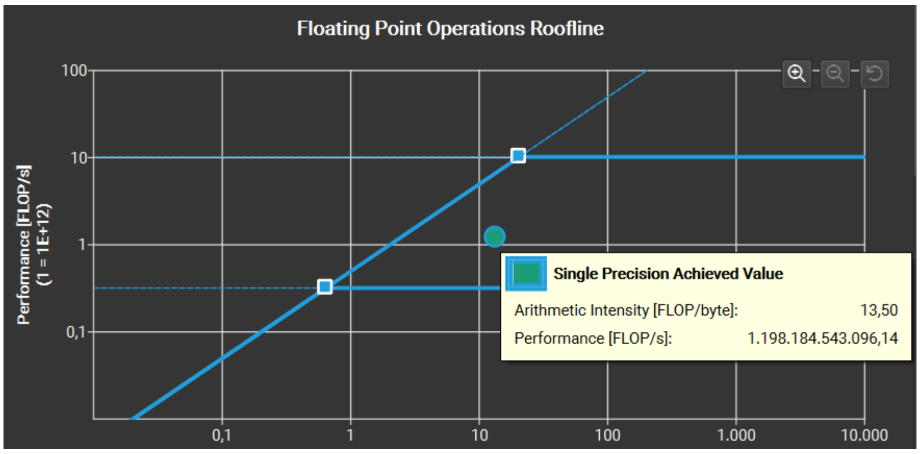
Table 1: Table 3. Throughput of Native Arithmetic Instructions.
(Number of Results per Clock Cycle per Multiprocessor)

Compute Capability	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A	256	128	2	256	128	256[3]		128	256
32-bit floating-point add, multiply, multiply-add	128		64	128		64		128		
64-bit floating-point add, multiply, multiply-add	4		32	4		32[5]	32	2		64
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	32		16	32		16				
32-bit integer add, extended-precision add, subtract, extended-precision subtract	128		64	128		64				
32-bit integer multiply, multiply-add, extended-precision multiply-add	Multiple instruct.					64[6]				
24-bit integer multiply (`__[u]mul24`)	Multiple instruct.									
32-bit integer shift	64		32	64						
compare, minimum, maximum	64		32	64						
32-bit integer bit reverse	64		32	64		16				
Bit field extract/insert	64		32	64		Multiple Instruct.			64	
32-bit bitwise AND, OR, XOR	128		64	128		64				
count of leading zeros, most significant non-sign bit	32 | 16 | 32 | 16									
population count	32		16	32		16				
warp shuffle	32					32[8]	32			
warp reduce	Multiple instruct.						16			
warp vote	64									
sum of absolute difference	64		32	64						
SIMD video instructions `vabsdiff2`	Multiple instruct.									
SIMD video instructions `vabsdiff4`	Multiple instruct.					64				
All other SIMD video instructions	Multiple instruct.									
Type conversions from 8-bit and 16-bit integer to 32-bit integer types	32		16	32		64				
Type conversions from and to 64-bit types	4		16	4		16[10]	16	2	2	16
All other type conversions	32		16	32		16				
16-bit DPX	Multiple instruct.									128
32-bit DPX	Multiple instruct.									64

144 Chapter 8. Performance Guidelines

Figure 66: The CUDA guide lists the number of results an SM can provide per cycle for each CC.

10.5 Floating Point Operations Roofline in Nvidia Nsight Compute

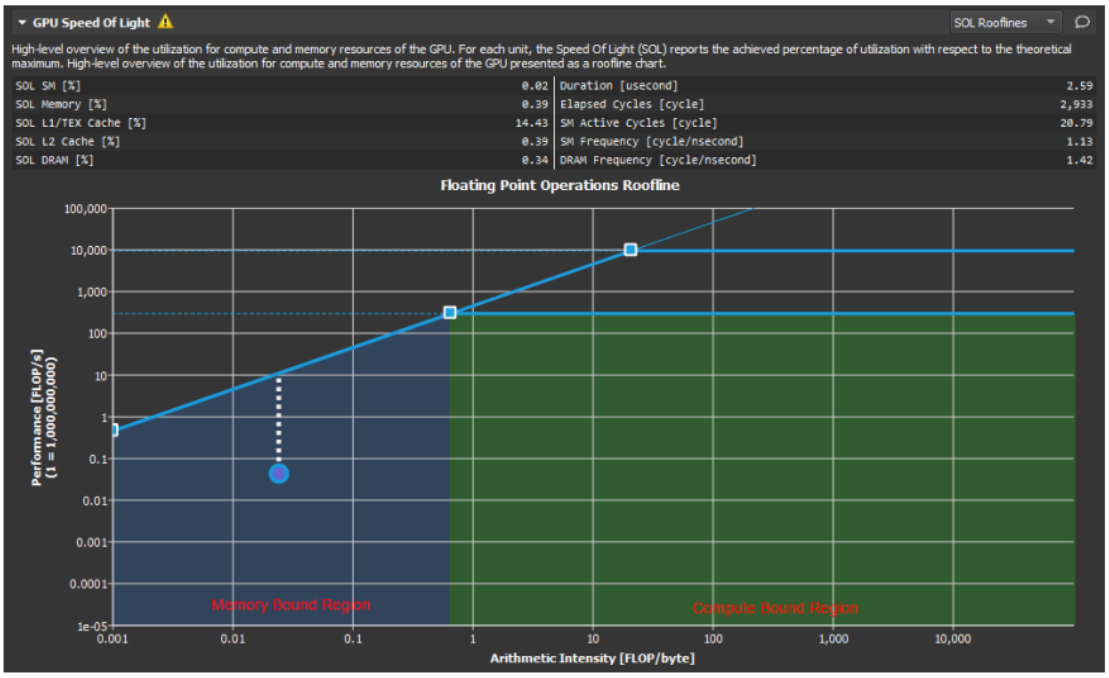


Single Precision Achieved Value

Arithmetic Intensity [FLOP/byte]:	13,50
Performance [FLOP/s]:	1.198.184.543.096,14

Figure 67: NNC provides a visualization of the achieved performance of a kernel, compared to the theoretical performance of the GPU the kernel runs on.

10.6 Floating Point Operations Roofline with illustrated regions



High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

SOL SM [%]	0.02	Duration [usecond]	2.59
SOL Memory [%]	0.39	Elapsed Cycles [cycle]	2,933
SOL L1/TEX Cache [%]	14.43	SM Active Cycles [cycle]	20.79
SOL L2 Cache [%]	0.39	SM Frequency [cycle/nsecond]	1.13
SOL DRAM [%]	0.34	DRAM Frequency [cycle/nsecond]	1.42

Figure 68: The *Kernel profiling Guide* illustrates the memory bound region and the compute bound region. The distance from the achieved performance (The blue circle) to the roofline boundary (The slanted blue line) is shown as a dotted white line. This distance represents potential performance improvement. If the blue circle is on top of the roofline, further improvements are only possible if the arithmetic intensity is increased at the same time.