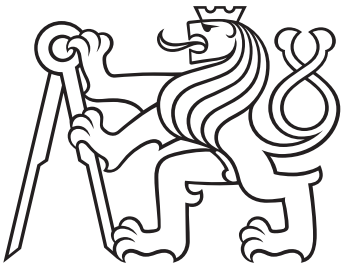
České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Obor: Aplikace softwarového inženýrství



Paralelní LU rozklad pro GPU

Parallel LU Decomposition for the GPU

VÝZKUMNÝ ÚKOL

Vypracoval: Bc. Lukáš Čejka
Vedoucí práce: doc. Ing. Tomáš Oberhuber, Ph.D.
Rok: 2022

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství

Akademický rok 2020/2021

ZADÁNÍ VÝZKUMNÉHO ÚKOLU

Student: Lukáš Čejka

Studijní program: Aplikace přírodních věd

Obor: Aplikace softwarového inženýrství

Název práce: Paralelní LU rozklad pro GPU

Pokyny pro vypracování:

1. Nastudujte LU rozklad a jeho využití pro řešení soustav lineárních rovnic.
2. Seznamte se s Croutovým algoritmem pro výpočet LU rozkladu a s jeho paralelní verzí.
3. Implementujte paralelní verzi LU rozkladu v CUDA pro GPU.
4. Aplikujte výsledný algoritmus na různé matice soustav a poměřte efekt urychlení konvergence.
5. Naměřte urychlení GPU verze LU rozkladu oproti CPU verzi.

Doporučená literatura:

[1] H. Anzt, T. Ribizel, G. Flegar, E. Chow a J. Dongarra. ParILUT - A Parallel Threshold ILU for GPUs. 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 2019. pp. 231-241. doi: 10.1109/IPDPS.2019.00033.

[2] Y. Saad. Iterative methods for sparse linear systems. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics, 2003. ISBN 0898715342.

[3] S. Bharatkumar, a H. Jaegeun. Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++. Packt Publishing, 2019. ISBN 1788996240.

Jméno a pracoviště vedoucího práce:

Ing. Tomáš Oberhuber, Ph.D.
Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Datum zadání výzkumného úkolu:

Termín odevzdání výzkumného úkolu:

V Praze dne

.....
vedoucí práce

.....
vedoucí katedry

Prohlášení

Prohlašuji, že jsem svůj výzkumný úkol vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Declaration

I declare that I have carried out my research project independently and I have used only the materials (literature, projects, software, etc.) listed in the bibliography.

V Praze dne ....

Bc. Lukáš Čejka

Poděkování

Chtěl bych poděkovat doc. Ing. Tomáši Oberhuberovi, Ph.D. za vedení mé práce a za podnětné návrhy, které ji obohatily. Poděkování patří také zpřístupnění výpočetní infrastruktury projektu financovaného OP VVV CZ.02.1.01/0.0/0.0/16_019/0000765 "Výzkumné centrum informatiky".

Acknowledgment

I would like to thank doc. Ing. Tomas Oberhuber, Ph.D. for supervising my project and for the inspiring proposals that enriched it. The access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" is also gratefully acknowledged.

Bc. Lukáš Čejka

Název práce:
Paralelní LU rozklad pro GPU

Autor: Bc. Lukáš Čejka

Studijní program: Aplikace přírodních věd
Obor: Aplikace softwarového inženýrství
Druh práce: Výzkumný úkol

Vedoucí práce: doc. Ing. Tomáš Oberhuber, Ph.D.
Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze
Konzultant: –

Abstrakt: Tento projekt se zaměřuje na paralelizaci metody LU dekompozice na grafických kartách (GPU). Mezi hlavni cíle této práce patří nastudování architektury a softwarové vrstvy GPU od společnosti Nvidia za účelem poskytnutí efektivního řešení rozkladu a porovnat jej s implementací na procesoru (CPU). Za tímto účelem byl implementován projekt obsahující funkcionální a výkonnostní testy. Výkonnostní testy byly spuštěny na sadě matic s různými vlastnostmi na moderním výpočetním systému. Výsledky výkonnostních testů byly analyzovány a následně byla doporučena implementace pro obecné použití.

Klíčová slova: LU dekompozice, Grafická karta (GPU), Compute Unified Device Architecture (CUDA), Nvidia, C++, Výkonnostní test, High Performance Computing (HPC), Porovnání výkonu, Porovnání přesnosti, Optimalizace, Template Numerical Library (TNL)

Title:
Parallel LU Decomposition for the GPU

Author: Bc. Lukáš Čejka

Abstract: The main focus of this project revolves around the parallelization of the LU decomposition method on the Graphics Processing Unit (GPU). Specifically, the study of the architecture and software layer of Nvidia GPUs in order to deliver an effective decomposition solution and compare it to the CPU implementation. To facilitate this, a project containing unit tests and a benchmarking framework was implemented. The benchmarks were run on a set of curated matrices with varying characteristics on a state-of-the-art compute cluster. The results of the benchmarks were analyzed and an implementation was recommended for general use.

Key words: LU decomposition, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA), Nvidia C++, Benchmark, High Performance Computing (HPC), Performance comparison, Accuracy comparison, Optimization, Template Numerical Library (TNL)

Contents

Introduction

The procedure of finding answers to questions that arise in scientific and engineering disciplines often begins by defining what the problem is. The next step is to formulate and present the problem in such a way that allows for a simple and accurate solution. One of the most common ways to represent a problem is by using a system of questions, for example, linear or partial differential equations. The former systems can be found in various fields such as engineering, physics, computer science, and economics. The latter systems regularly emerge in both physics and engineering, for example, electrodynamics, fluid dynamics, thermodynamics, etc. Apart from emerging in similar fields, these two systems of equations can both be represented by matrices and thus, solved using basic tools of linear algebra.

Consequently, from the perspective of computer science, such problems can be solved using many different methods, for example, direct methods. This group of methods is characterized by finding the solution to a problem by a finite sequence of operations. While such methods can - theoretically - provide an exact solution, in reality, due to rounding errors, their accuracy may not always be adequate. Furthermore, owing to the methods' sequential nature, they can seldom be parallelized and as such, they are often run on a Central Processing Unit (CPU) which can take a considerable amount of time. For these reasons, an alternative means of arriving at a solution can be found in iterative methods which use an initial value as a starting point and then iteratively converge to an approximate solution. One of the advantages of using iterative methods over direct methods is that their procedures can usually be parallelized. This, combined with the rise in popularity of utilizing Graphics Processing Units (GPUs) for scientific computations, led to the tailoring of parallelizable algorithms for optimal execution times on the GPU. This project aims to repurpose a direct method that can be used to solve systems of equations (LU decomposition) into an iterative method and to parallelize its execution on the GPU.

The first chapter presents a detailed description of the hardware of GPUs along with the software that is used to leverage their computational power. Additionally, the LU decomposition method is introduced. The second chapter describes the implementation of the project that houses the direct and iterative versions of the LU decomposition method and any accompanying functionalities. Finally, the last chapter presents the results of benchmarks that were run in order to compare the CPU and GPU implementations.

Chapter 1

Theory

This chapter will present the core theory of areas used in this research assignment. Firstly, Graphical Processing Units (GPUs) will be introduced with an emphasis on the architecture of GPUs (only Nvidia GPUs as the Nvidia API, CUDA, was used to develop this project). Then, CUDA will be described in greater detail. Finally, the theory behind Lower-Upper (LU) Decomposition will be presented.

1.1 GPUs

From an average consumer's perspective, a Graphics Processing Unit (GPU) is a component of a computing system that performs graphical operations, for example, processing images. For the sake of the example, let image processing be simplified into the following: images are composed of pixels and each pixel needs to be processed. Since a GPU contains thousands of processing units it can perform a large number of operations in parallel, for example, processing a large number of pixels at once, thus making it highly efficient at image processing. While this has a wide range of uses, there is another field where the highly parallel nature of the GPU can be utilized: General-Purpose Computing on Graphics Processing Units (GPGPU).

1.1.1 General-Purpose Computing on Graphics Processing Units (GPGPU)

As described in Section 1.2 of the author's bachelor thesis *Formats for storage of sparse matrices on GPU* [1], GPUs started to evolve at the turn of the 21st century from purely graphics-processing units into devices capable of general-purpose computing, i.e. computation that is not necessarily related to graphics. The significance of this event lies in the use of graphics cards for seemingly any parallelizable computational task that would benefit from the abundance of slower processing units on a GPU rather than fewer faster processing units found on a CPU [2].

For reference, high-end desktop CPUs today usually have anywhere from 8 to 16 cores, while server CPUs can have upwards of 64 cores (not counting hyper-threading and similar technologies). In general, CPUs with more cores have a lower clock speed - see Figure 1.1 for a selection of processors along with their core count and clock speeds [3].

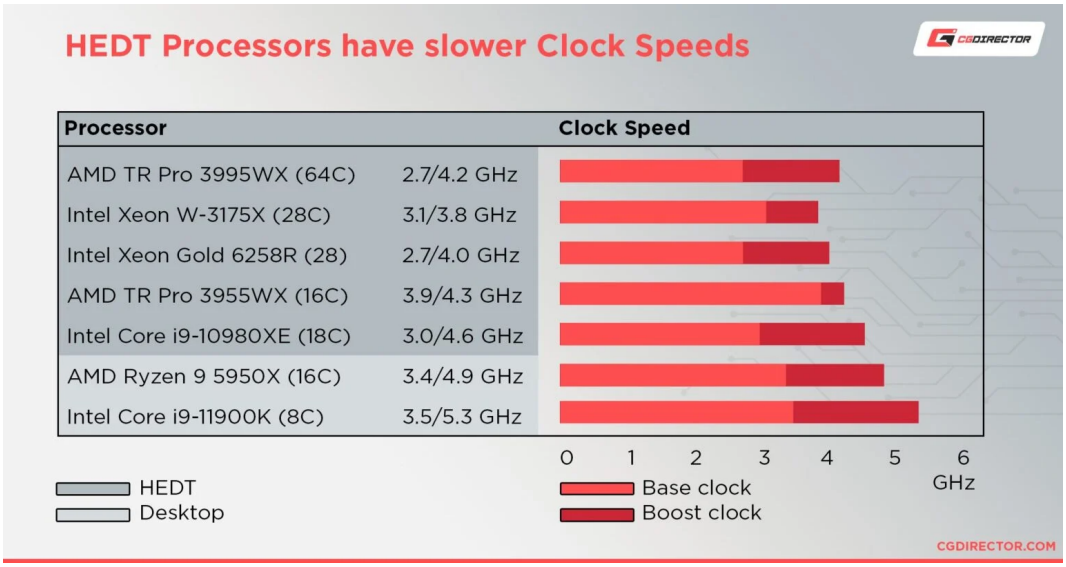


Figure 1.1: Selection of desktop and server CPUs - including the number of cores and clock speed [3]. HEDT stands for High-End Desktop Processor - in this instance, it also includes CPUs predominantly used in servers.

While CPUs are widely used for sequential tasks, they can also be used for tasks that can be described as parallelizable, for example, processing of requests to a server. They are not suitable for highly parallel tasks such as image processing, where GPUs excel due to their architecture. Figure 1.2 shows the comparison of a CPU and GPU architecture. The example CPU in the figure has - among other components - 4 cores each having its own L1 cache and controlling component. This configuration allows for executing a thread (series of operations) at a high speed and lower throughput (fewer threads running at once). Simply, the CPU is more suitable for the rapid completion of serialized instructions.

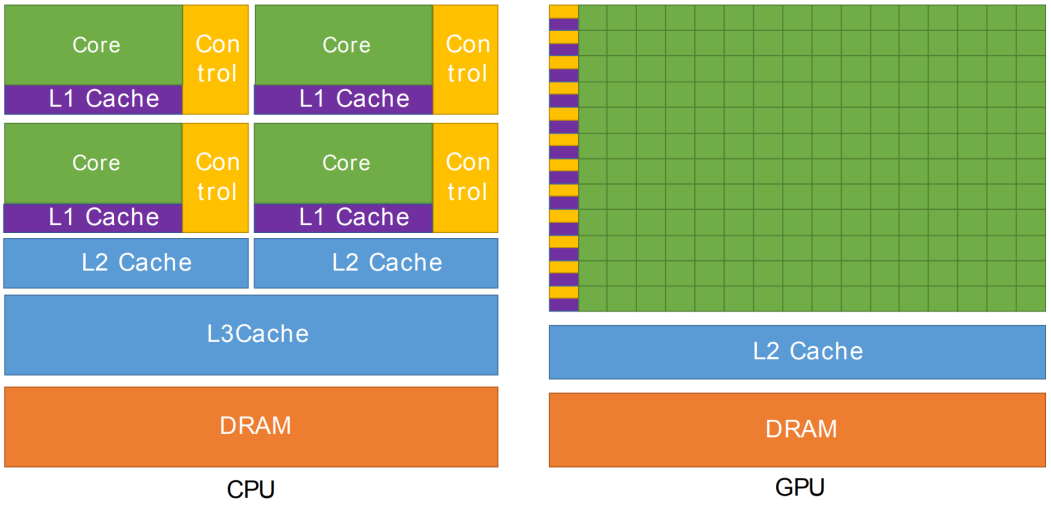


Figure 1.2: Comparison of the architecture of CPUs and GPUs. On one hand, CPUs have fewer cores, and more complex controlling logic compared to GPUs, and the cores of a CPU have a higher clock speed. On the other hand, GPUs have a much higher number of cores clocked at lower speeds. Taken from Nvidia's *CUDA C++ programming guide* [2].

Nvidia GPUs, on the other hand, are engineered to have many smaller controlling units called SMs (Stream Multiprocessors) that aim to schedule and task individual processing units found on the GPU. Furthermore, in Figure 1.2 it can be seen that GPUs have a different cache structure. Specifically, GPUs only have two levels, whereas CPUs have three. In summary, the GPU has more transistors for processing data - namely operations that use floating-point computations. Moreover, the architecture of the GPU allows it to compensate for memory access delays by performing computations simultaneously [2].

A prime example of where this characteristic gives a significant advantage to the GPU over the CPU is during matrix multiplication which will be described later in Section 1.2.6.

For completion, see Table 1.1 for a selection of GPUs along with their specifications.

Nvidia	GTX 1070	RTX 3060	V100	A100
GPU	GP104 (Pascal)	GA106 (Ampere)	GV100 (Volta)	GA100 (Ampere)
SMs	15	28	80	108
TPCs	15	14	40	54
FP32 Cores / SM	NA	NA	64	64
FP32 Cores / GPU	NA	NA	5,120	6,912
FP64 Cores / SM	NA	NA	32	
FP64 Cores / GPU (excl. Tensor)	NA	NA	2,560	3,456
CUDA Cores / SM	128	128	NA	NA
CUDA Cores / GPU	1,920	3,584	NA	NA
Tensor Cores / SM	NA	4	8	16
Tensor Cores / GPU	NA	112	640	432
GPU Boost Clock	1683 MHz	1777 MHz	1530 MHz	1410 MHz
Peak FP32 TFLOPS	6.5	12.7	15.7	19.5
Peak FP64 TFLOPS	0.202	0.199	7.8	9.7
Peak Tensor TFLOPS	NA	51	125	312/624
Memory Bandwidth	256 GB/s	360 GB/s	900 GB/s	1555 GB/s
Texture Units	120	112	320	432
Memory Interface	256-bit	192-bit	4096-bit HBM2	5120-bit HBM2
Memory Size	8 GB	12 GB	32 GB	80 GB
L2 Cache Size	2048 KB	3072 KB	6144 KB	40960 KB
Shared Memory Size / SM	96 KB	128 KB	Configurable up to 96 KB	Configurable up to 164 KB
Register File Size / SM	256 KB	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	7168 KB	20480 KB	27648 KB
TDP	150 Watts	170 Watts	300 Watts	400 Watts
Transistors	7.1 billion	12 billion	21.1 billion	54.2 billion
GPU Die Size	314 mm^2	276 mm^2	815 mm^2	826 mm^2
Manufacturing Process	TSMC 16nm	Samsung 8N	12 nm FFN	7 nm N7

Table 1.1: Comparison of GPUs: GTX 1070 (Turing architecture), RTX 3060 (Ampere architecture), V100 (Volta architecture), A100 (Ampere architecture). The GPUs in this table assume the best possible configuration of their respective card, for example, the version with the most possible VRAM. Features that are important when it comes to card performance have been denoted in green. The data was obtained from various sources for the GTX 1070 [4, 5, 6, 7] the RTX 3060 [8, 9, 10, 11], the V100 [12] and the A100 [13, 14].

The table shows specifications for two different categories of GPUs: consumer and professional (commercial). Consumer cards (in the table: GTX 1070, RTX 3060) can be found in regular desktop PCs and are intended more for video gaming, rather than raw computing power. However, these cards are mentioned as the optimization of this project's implementation was done on machines that included them. On the other hand, professional cards are intended mainly for GPGPU, or - in some specific cases - for machine learning.
Characteristics that can be used to compare GPUs are, for example, peak TFLOPS (TFLOPS - how many trillion floating-point operations can the processor perform per second), memory bandwidth, and TDP. The value that separates commercial cards from consumer cards is the peak FP64 (double precision) TFLOPS, which is the highest for the A100 at 9.7 TFLOPS, with the V100 being slightly slower at 7.8 TFLOPS, whereas the consumer cards perform significantly worse at around 0.2 TFLOPS.
In summary, professional cards are heavily preferred when it comes to GPGPU, however, consumer cards can be and are used for development as they are more affordable.

1.2 Compute Unified Device Architecture (CUDA)

The Compute Unified Device Architecture (CUDA) is a programming model, sometimes referred to as a parallel computing platform, introduced by Nvidia in 2006 [15]. It is designed to give developers low-level access to GPU hardware, for example, fine-tuning the assignment of processing units or memory allocation, thus, allowing them to utilize the full potential of GPUs and tailor their use for specific applications. CUDA supports a variety of programming languages, for example, C++ (used in this project) and Fortran, however, adaptations for other languages such as Python, Perl, Java, Ruby, MATLAB, Julia, etc. have been created [16].

This section will first explain some basic terminology followed by the theory behind CUDA from the simplest concept - a thread - to how threads are managed. Then, memory will be introduced similarly: from per-thread local memory to global memory. Subsequently, asynchronous concurrent execution will be described. Finally, basic CUDA extensions of the C++ language will be presented and their translation into the thread and memory management systems will be shown on a simple example program.

1.2.1 Introductory Terminology

CUDA introduces many unique concepts that come with their own naming scheme. This section will list and explain some basic terminology that will be used throughout the project [17]:

- **Host** - Central Processing Unit (CPU) and its memory. The host provides data to the GPU and instructs it to execute various instructions.
- **Device** - Graphics Processing Unit (GPU) and its memory. The device executes specialized instructions provided by the host.
- **Kernel** - the term used for a special type of function that, unlike regular functions, can only be called from the host and executed on the device. Furthermore, when a CUDA kernel is called it can be executed in parallel across several threads.

1.2.2 Thread Management

This section will aim to describe how CUDA handles its basic execution units: threads. First, a thread itself will be defined and described followed by explanations of multiple encompassing thread structures.

CUDA thread The thread management system within CUDA begins, at the most basic level, with its smallest execution unit, a thread. According to *CUDA C++ Programming Guide*, a thread is an executed sequence of operations [2]. Due to the highly parallel nature of the GPU - spreading the workload across thousands of execution units - a CUDA thread is designed to be lightweight (unlike a typical CPU thread - assuming no hyper-threading is present). In this context, "lightweight" signifies that the GPU is capable of easily switching between threads. An example showing how a set of instructions can be run on a set of 8 threads can be seen in Figure 1.3.
In this project, the terms "CUDA thread" and "thread" will be used interchangeably. For a more detailed explanation of the difference between a CUDA thread and that of the CPU see *Formats for storage of sparse matrices on GPU* [1].

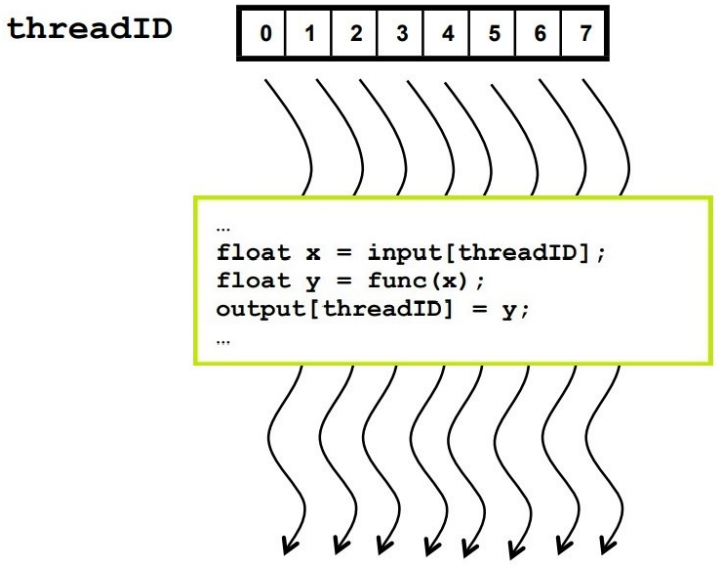


Figure 1.3: On an Nvidia GPU, eight threads with IDs from `0` to `7` execute the code in this example. A value from the array `input` with `threadID` as the index (i.e. each thread will read a value at a different index) is changed using the function `func(x)` and then stored in the same order as before into the array `output`. Taken from *Format for storage of sparse matrices on GPU* [1, 17].

Warp To execute thousands of threads simultaneously, SMs of an Nvidia GPU use the Single-Instruction Multiple-Thread (SIMT) execution model. The abbreviation SIMT comes from the amalgamation of SIMD (Single-Instruction Multiple-Data) and multi-threading (executing multiple sequences of instructions simultaneously). Specifically, this approach consists of multiple threads executing the same computations on different data items [18]. In terms of CUDA, the core of the SIMT architecture is a so-called "warp" that is made up of 32 threads. In other words, the smallest number of threads that can be executed simultaneously is 32. If fewer threads are required, for example, only one thread, then CUDA will still allocate 32 threads with 31 being inactive. Similarly, if some threads of a warp

are to execute different instructions (e.g., as a result of a conditional statement) then the different sets of instructions are executed serially across their respective threads - this issue is known as "thread divergence". An example of thread divergence can be seen in Figure 1.4; the warp in the example is simplified for the explanation and as such contains only 8 threads.

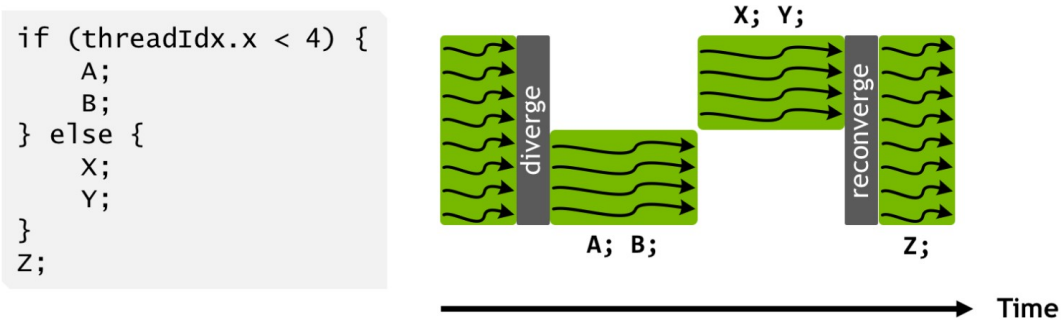


Figure 1.4: Conditional code execution by a warp of 8 threads; each thread is represented by one wavy line. Due to the condition, threads with an ID between `0` and `3` will only execute statements `A`, `B`, and `Z`. However, simultaneously, threads with IDs between `4` and `7` will be put on hold. Comparably, while threads with IDs between `4` and `7` will be executing statements `X` and `Y`, the remaining threads will be idle. Thus, thread divergence has occurred within the 8-thread warp. Taken from Nvidia's developer blog post: *Inside Volta: The World's Most Advanced Data Center GPU* [19].

Block It is not always necessary to use such granularity that warps can provide. Developers can choose to execute code on a group of up to 1024 threads (with CUDA Compute Capability greater than 3.5) - referred to as a block. Threads in a block can cooperate via shared memory (subsection 1.2.3 will explain more) and they can be synchronized - this is not possible for threads in different blocks. Threads in a block can be structured in 1, 2, or 3 dimensions (x, y, z); each thread has a unique ID in every dimension. The 1024 threads-per-block limit encompasses all 3 possible dimensions, in other words, the total number of threads must be at most 1024 across all dimensions (`num_threads_x_dim` $\cdot$ `num_threads_y_dim` $\cdot$ `num_threads_z_dim` ≤ 1024) [20, 2].

Grid In terms of CUDA, a grid consists of multiple blocks of threads. Similarly to how threads can be structured within blocks, blocks can be structured within grids - up to 3 dimensions of blocks; each block has a unique ID in every dimension. The maximum number of blocks in every dimension is set to $2^{31} - 1$ (65 536). One of the main reasons for having another structure on top of blocks (apart from memory management - detailed explanation in Section 1.2.3) stems from the fact that while the limit for threads per block is set to 1024, GPUs are capable of running a multitude more threads in parallel. This means that the grid structure allows for the same code to be executed simultaneously on a group of thread blocks. However, it is important to note that there is a limit to how many threads can be active at once. When this limit is reached, blocks of threads are executed sequentially in such a way that allows for near-maximal concurrently active threads. Another noteworthy aspect is the difference between the terms "number of allocated threads" and "number of active threads". The former means how many threads are allocated in total, i.e. both active (currently executing) and inactive (scheduled) threads, whereas the latter refers to threads that are actively being executed at a point in time.
A visualization of CUDA's thread structuring can be seen in Figure 1.5.

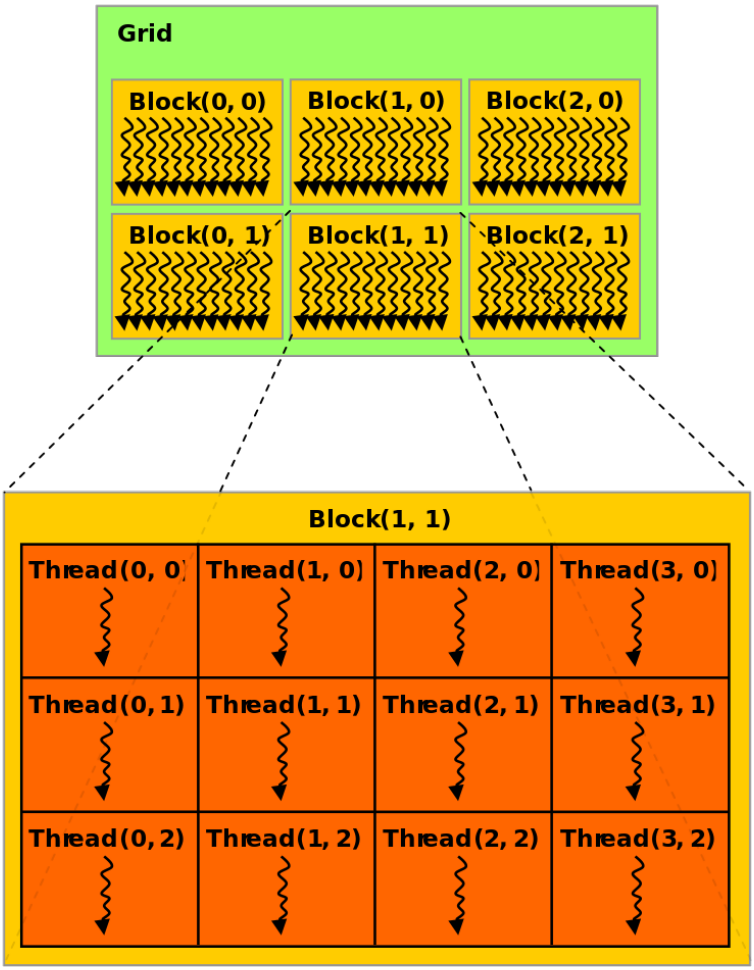


Figure 1.5: CUDA thread structuring of a 2D grid made up of 2D blocks of threads. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

1.2.3 Memory Management

This section aims is to present how memory management works in CUDA. Different types of memory available on the GPU will be presented, ranging from per-thread local to global memory.

Per-thread local memory When a kernel is executed on the GPU across several threads every thread has its unique ID stored in a variable that is available within the kernel. This variable is stored in memory which is referred to as *per-thread local memory*. In other words, every thread has its own local memory where it can store variables. It is important to note that per-thread local memory consists of two different types - registers and local memory. Registers is on-chip memory (low latency and high bandwidth), however, it is limited in capacity at 255 32-bit registers per thread in CUDA Compute Capability 3.5 and higher, which does not allow for extensive kernels containing many variables. The compiler will store almost all variables allocated within the kernel into registers, except for [2]:

- Arrays indexed with constant quantities
- Large structures or arrays that would use too much registers space
- Any variable if the kernel uses more registers than available - called *register spilling*

The variables and structures mentioned above that the compiler will not store in registers will instead be stored in purely local memory which resides in device memory. Device memory is also referred to as global memory which is off-chip and therefore, any accesses by threads into their purely local memory will be slower than accessing registers. Figure 1.6 shows available accesses of threads to different types of memory.
As of CUDA Compute Capability 3.5, the amount of purely local memory available for every thread is only 512 KB [2]. In summary, per-thread local memory is often used sparingly to avoid register spilling and the accompanied slower memory access, however, if used efficiently, it can be helpful when complex indexing is required for some calculations.

Shared memory The next layer of memory is per-block memory referred to as *shared memory*. As the name suggests, this memory is shared by all threads belonging to a particular block. Similarly to registers, shared memory is on-chip and therefore it is fast, however, it is also limited in capacity depending on the CUDA Compute Capability version - see table 1.2 for specific values.

Compute Capability	3.5 - 6.2	7.0 - 7.2	7.5	8.0	8.6	8.7
Max. shared memory per block [KB]	48	96	64	163	99	163

Table 1.2: Maximum shared memory per thread block across different CUDA Compute Capabilities. Nvidia notes in their documentation that any value above 48 KB requires the use of dynamic shared memory. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Shared memory can either be allocated statically or dynamically. The former needs the size of allotted memory to be known at compile time, for example, an array of constant size denoted within the kernel using a dedicated prefix that indicates it is meant to be stored in the shared memory of each thread's block. On the other hand, the latter - dynamic allocation of shared memory - allows for the size of required memory to be determined at runtime, however, this means that the developer must specify the size of the shared memory as one of the parameters when calling the kernel.

The shared memory of each block is divided into 32-bit (4-byte) memory modules called words. A single word can store a float, half a double, 32-bit int, etc. - anything that can be stored in 32 bits. Apart from words, there are 32 banks per block. The number of words each bank contains depends on the size of the shared memory allocated. Since the size of shared memory can be up to 48 KB or more - depending on the version of CUDA Compute Capability - then the number of words per bank can differ between versions. Figure 1.7 visualizes the division of shared memory into banks and words.

Manipulating memory in banks can be fast and efficient under certain conditions. Let us consider the following example: Figure 1.7b is the shared memory of a block that is used in a kernel. In this kernel, there is an array made up of 256 floats (32-bit) - each word in the figure is a single float. Furthermore, the IDs of words in the figure correspond to indices of floats in the array, thus, float values stored next to each other in memory belong to successive banks, for example, `arr[0]` belongs to `Bank0`, `arr[1]` belongs to `Bank1`, `arr[32]` belongs to `Bank0`, etc.

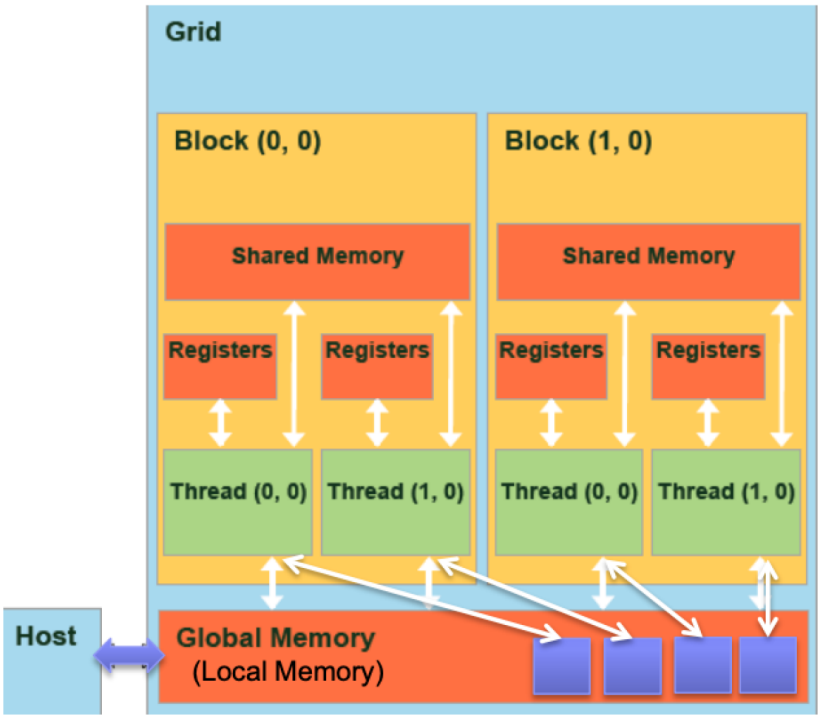


Figure 1.6: Different CUDA memory types visualized. Each thread has access to its local memory comprised of registers, local memory (blue squares in global memory), shared memory of its block, and global memory. Taken from Yao Hsiao's *GPU - CUDA introduction* [21].

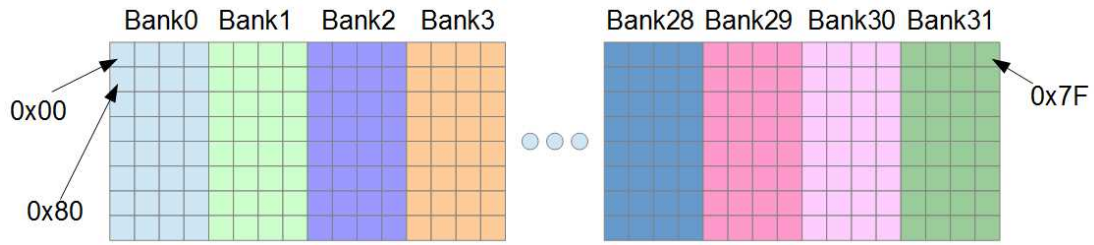
As mentioned by Chris Rose in *CUDA Succinctly* [22], this division of words into banks is salient as each bank can serve only one word to a warp at once. This means that if a single word is requested from each bank, then all 32 threads of a warp can be served by all 32 banks simultaneously and expeditiously. Data serving is also fast when the same word from shared memory is accessed by all threads of a warp; this is called a *broadcast* as shared memory is read once and the value is sent to all threads of the warp at once.

A concept similar to broadcasting is called a *multicast*, which is an operation that is used when the same word from any particular bank is accessed by more than one thread of a warp. In this case, the threads that accessed the word will all receive it simultaneously from its bank. Under good conditions, the multicast operation is as fast as a broadcast. It is important to note that multicast is only available for compute capability 2.0 and higher.

However, the operations above all assumed good conditions for reading and writing data. The primary originator of bad conditions for accessing shared memory is called a *bank conflict*. Bank conflict describes an instance when threads of a warp request more than one word from any single bank. When this occurs, reading and writing of a word will be performed serially by the bank. For example - looking at Figure 1.7b - let thread `0` and thread `1` access words `0` and `32` respectively. Since both words belong to `Bank0`, then a bank conflict has occurred and the bank cannot distribute words' data in parallel. Instead, `Bank0` will first serve thread `0` with word `0` and then thread `1` with word `32`.

To facilitate further understanding, examples from *Cuda succinctly* [22] and *CUDA C++ Programming Guide* [2] are included below.

Firstly, Table 1.3 presents different shared memory access patterns - including notes on the speed of execution. Then, Figure 1.8 portrays shared memory access using random permutations, multicast, and



(a) Shared memory divided into banks (columns under each `Bank*`), words (4 same-color squares of a row), and bytes (individual squares). In the first row, the first 4 bytes under `Bank0` - beginning at address `0x00` - make up the `0th` word (as shown in Figure1.7b). The four bytes to the right of it (first row under `Bank1`) make up the `1st` word. Memory addresses of some bytes are shown as hexadecimal.

(b) Shared memory is divided into banks (column under each `Bank*`) and words (individual rectangles with IDs inside).

Figure 1.7: Illustration of shared memory. In this example, there are 32 banks; each one has eight 4-byte (32-bit) words - in Figure 1.7a one byte is one square. Reading and writing into the `0th`, `32nd`, `64th`, etc. word is the responsibility of `Bank0`; reading and writing into the `1st`, `33rd`, `65th`, etc. word is the responsibility of `Bank1` and so on. Taken from *Chapter 6: Shared Memory* of *Cuda Succinctly* [22].

broadcast. Finally, Figure 1.9 shows examples of strided shared memory access with and without bank conflict - additionally, this figure also visualizes some examples from Table 1.3.

In summary, ensuring that bank conflicts do not occur is pivotal to achieving efficient use of shared memory and subsequently high bandwidth.

Global memory The final, all-accessible memory layer is called *global memory*. Some sources also refer to it as *device memory* since global memory resides in the device's DRAM (Dynamic Random Access Memory) - not to be confused with the host's memory, often referred to as, RAM. The modifier "global" represents the all-accessible aspect of this type of memory, as it can be accessed by the host and the device, thus, serving as a memory communication medium that can be used to transfer data between the two, and house input and output data for kernels [23]. Global memory is available for all threads on the device, regardless of what thread structure they belong to. In other words, all grids have access to the same global memory - shown in Figure 1.10.

Unlike shared memory, global memory functions similarly to RAM: it is initialized with the startup of the program, it will be available while the program is running, and it is terminated with the termination

Access Pattern	Notes
`arr[0]`	Fast - broadcast to all threads of the grid
`arr[bID]`	Fast - broadcast to all threads of a block
`arr[tID]`	Fast - all threads request from different banks
`arr[tID/2]`	Fast - multicast; every two threads read from the same bank
`arr[tID*2]`	Slow - two-way bank conflict
`arr[tID*3]`	Fast - all threads access different banks
`arr[tID*32]`	Egregiously slow - 32-way bank conflict

Table 1.3: Access patterns to shared memory. `arr` is an array stored in the shared memory of a thread block. `tID` is the ID of a thread and `bID` is the ID of a block. Taken from *Chapter 6: Shared Memory* of *Cuda Succinctly* [22].

of the program. To use global memory, the developer must manually allocate memory on the device, then copy the data from the host to the device, and finally, deallocate (free) the data from the device [23] once it is not required.
As mentioned above, in Paragraph *Per-thread local memory* in Section 1.2.3, for this project global memory is assumed to be off-chip - for completion, an exception to this is mentioned by Mark Harris in *How to Access Global Memory Efficiently in CUDA C/C++ Kernels* [23]: "*Depending on the compute capability of the device, global memory may or may not be cached on the chip.*". Accessing (reading and writing) global memory is slower than accessing the aforementioned types of memory. To minimize this bottleneck, Nvidia advises keeping the number of transactions that require accessing global memory to as few as possible [23]. One of the main concepts that achieves this is called *global memory coalesced access*. This methodology takes full advantage of the SIMD approach, specifically, instruction execution by threads in warps. In other words, it uses the fact that threads in a warp execute the same instruction simultaneously - in this instance: combine multiple memory accesses into a single transaction.
This means that upon the execution of a load instruction by all threads in a warp, the device will detect whether the threads are attempting to access successive global memory locations. If the accessed addresses are successive, then the accesses are coalesced (amalgamated) by the device into a consolidated access to consecutive DRAM locations [24].
More specifically, a single coalesced access into global memory - composed of a single access instruction performed by threads of a warp - occurs only if the following conditions are satisfied [25, 2]:

1. Each thread accesses a memory element of size 4, 8, or 16 bytes
2. The device memory is accessed by memory transactions of 32, 64, or 128 bytes
3. Each segment must be aligned to its size - the first address is a multiple of their size

The alignment requirement signifies that accessing (reading and writing) words of size 1, 2, 4, 8, or 16 bytes within global memory instructions is supported. Moreover, if the size of data stored in global memory is 1, 2, 4, 8, or 16 bytes and it is aligned, then access to this data is joined into one memory transaction.

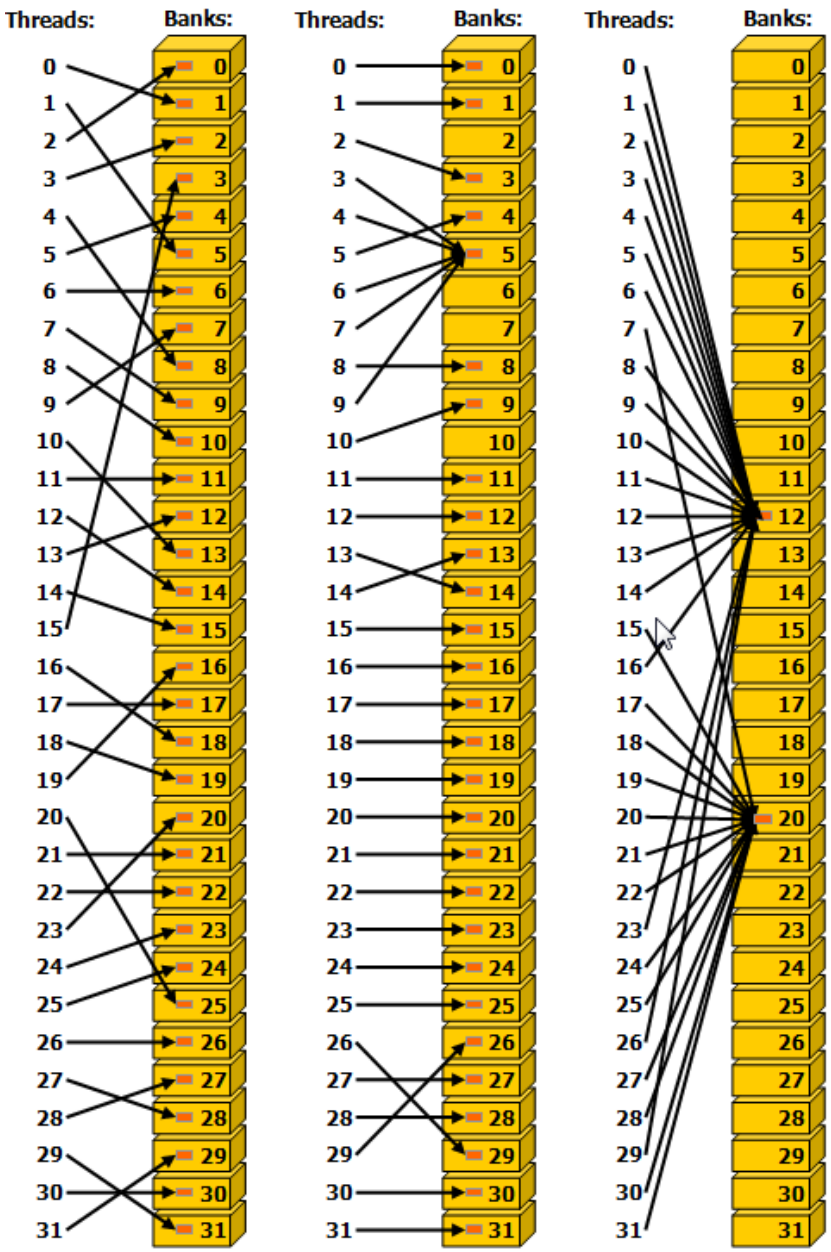


Figure 1.8: Irregular shared memory accesses on 3 separate examples; yellow rectangular cuboids are banks; small orange rectangles are words. The left sub-image shows threads of a warp accessing shared memory randomly, however, in such a way that does not cause bank conflicts. The middle sub-image shows shared memory accessed by threads using the multicast operation, specifically, threads 3, 4, 5, 6, 7, and 9 accessing the same word from bank 5; other threads all access one word any bank each - no bank conflict. The right sub-image shows threads accessing shared memory via two broadcast operations. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Ad point 2 of the conditions above: when threads of a warp access words larger than 4 bytes, then the memory request made by the warp is divided into separate 128-byte memory requests. These newly-formed requests are subsequently performed separately, based on the word size [2]:

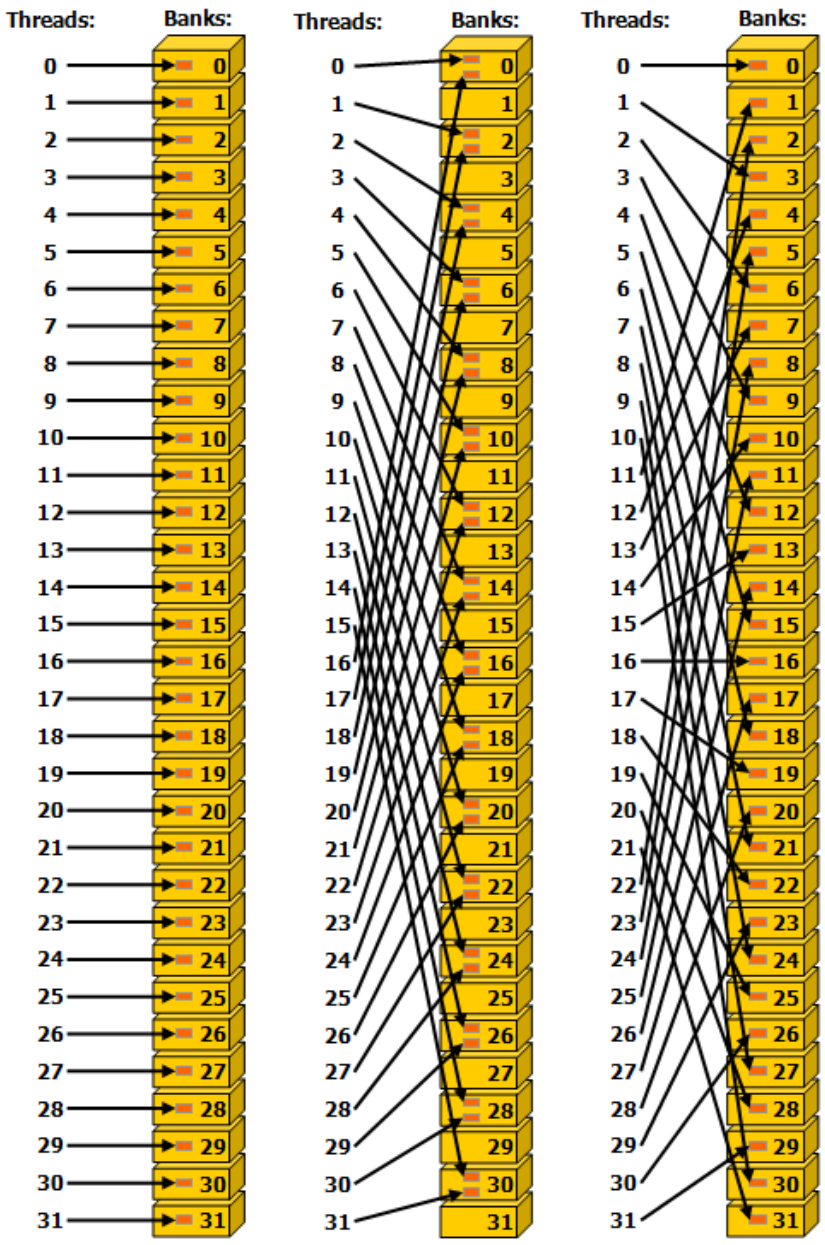


Figure 1.9: Shared memory access using striding on 3 separate examples; yellow rectangular cuboids are banks; small orange rectangles are words. The left sub-image shows threads of a warp accessing shared memory with a stride of 1 (no bank conflict). The middle sub-image shows shared memory accessed by threads with a stride of 2 (equivalent to `arr[tID*2]` from Table 1.3; two-way bank conflict as threads access two different words from the same bank). The right sub-image shows threads accessing shared memory with a stride of 3 (equivalent to `arr[tID*3]` from Table 1.3; no bank conflict). Taken from Nvidia's *CUDA C++ Programming Guide* [2].

- 8 bytes - 2 memory requests (1 for each half-warp)
- 6 bytes - 4 memory requests (1 for each quarter-warp)

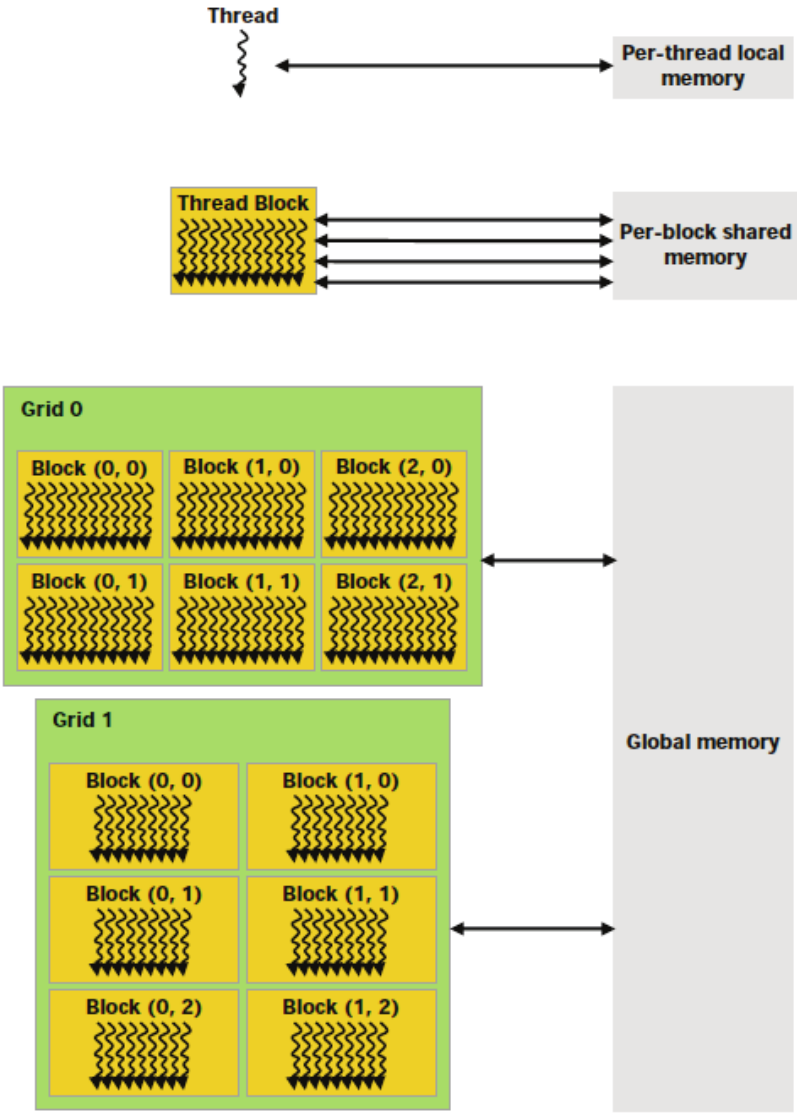


Figure 1.10: CUDA memory structuring. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Table 1.4 shows examples of built-in vector types that are aligned in global memory by default; the entire list can be found in Nvidia's *CUDA C++ Programming Guide* [2].

The size and alignment of other types, such as structures, must be approached manually: forcing the compiler to align them by using specifiers in code. For example, structures of 8 or 16 bytes that are to be aligned are declared using `__align__(8)` and `__align__(16)` respectively - shown in Listing 1.1.

```
1 struct __align__(8) {
2   float x;
3   float y;
4 };
5
6 struct __align__(16) {
7   float x;
8   float y;
```

Type	Alignment
int1, int2, int3, int4	4, 8, 4, 16
long1, long2, long3, long4	8, 16, 8, 16
float1, float2, float3, float4	4, 8, 4, 16
double1, double2, double3, double4	8, 16, 8, 16

Table 1.4: Built-in vector types that are aligned by default. The suffix numbers specify the number of components the vector composes of, for example, a vector of type `int2` has two components: `(x, y)`. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

```
9   float z;
10 };
```

Listing 1.1: Declaration of to-be-aligned 8- and 16-byte structures. The lower example is aligned to 16 bytes as 12-byte alignment (3 4-byte floats) does not coalesce, therefore, 4 bytes are used for padding. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

On the other hand, if 8-byte or 16-byte words are not aligned, then reading them yields results offset by some words.
However, ultimately, if the size and alignment requirements are not adhered to, then the access to global memory is compiled into multiple instructions in such a way that they will not fully coalesce, which leads to lower bandwidth and suboptimal performance.
Examples of coalesced and non-coalesced memory accesses by a 16-thread warp are shown in Figures 1.11 and 1.12.

In summary, coalesced access (reading and writing) to global memory must be fulfilled to not lower bandwidth - especially when dealing with custom non-built-in vector types - as global memory is implicitly high-latency and low-bandwidth compared to other memory types mentioned above.

Page-locked host memory This type of memory that can be utilized within CUDA is different from the previous types in that it is not located on the device. Page-locked host memory - sometimes referred to as pinned - resides in the memory of the host. The difference between it and the host's regular pageable memory is its permanent placement and other properties related to CUDA. Firstly, it can be allocated and deallocated using functions that come with CUDA: `cudaHostAlloc()` and `cudaFreeHost()`, or, if the memory on the host has already been allocated using `malloc()` it can be registered - made available within CUDA - using `cudaHostRegister()`. According to *CUDA C++ Programming Guide* [2], there are many advantages to using page-locked host memory:

- Concurrent data transfer between page-locked host memory and device memory, and kernel execution - so-called *asynchronous concurrent execution* (detailed in Section 1.2.4).
- Elimination of data copying between the host and device by mapping page-locked host memory to the device's address space - so-called *mapped memory* (detailed below).

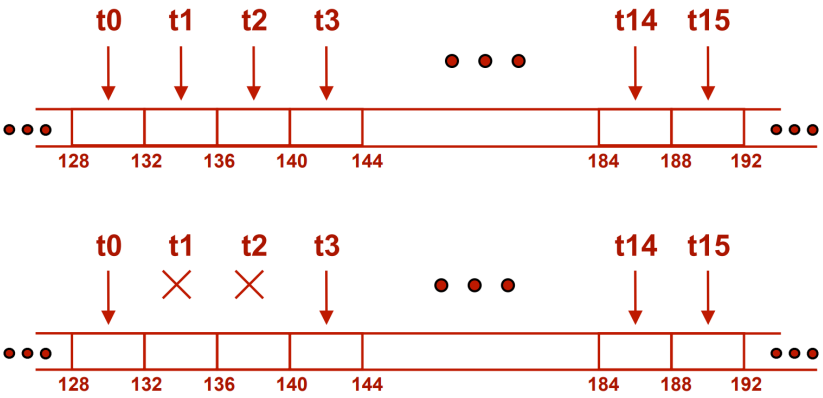
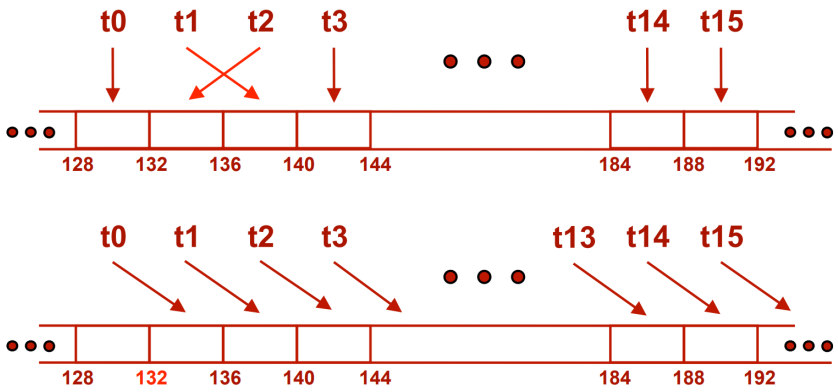


Figure 1.11: Coalesced access to global memory. Threads are denoted with `t<ID>` and the numbers below the rectangular memory locations are global memory addresses. In the upper image, all threads of the warp access successive memory addresses that house words of size 4 bytes (`float`), the size of the memory transaction is 64 bytes, and the segment is aligned to its size so that the first address is a multiple of its size: segment is 64 bytes and the starting address is 128: $128/64 = 2$. The lower image shows an example where two threads from the 16-thread warp do not access data. Thus, threads of the warp have two separate execution paths (thread/warp divergence) that are executed sequentially. However, from a memory standpoint, the access still coalesces as all requirements are fulfilled. Taken from Martínez Manuel Ujaldón's *CUDA Optimizations, Debugging and Profiling* [25].

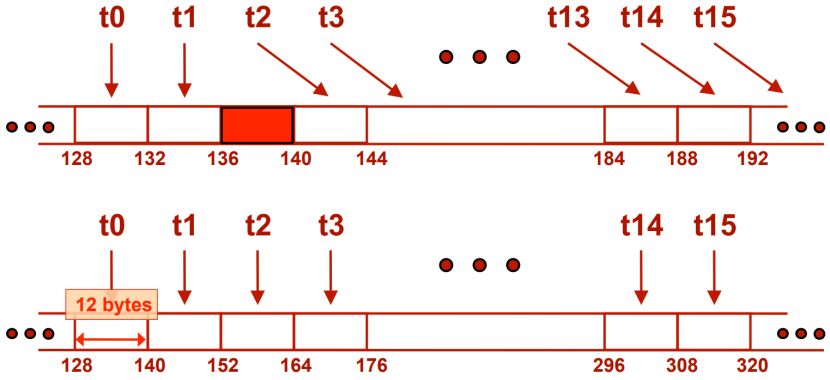
- Higher bandwidth when copying between page-locked host memory and device memory on systems with a front-side bus. Furthermore, if the page-locked host memory is allocated using the write-combining principle, then bandwidth can be even higher.

Along with the above-mentioned advantages, there are drawbacks such as the small size of page-locked host memory. Since it is often used by the operating system for paging (host stores data - to-be-used in main memory - in paged memory on a drive rather than in RAM) it is not an abundant resource. Therefore, incautious use can lead to allocation failures and reduced system performance [2].

Mapped memory As mentioned above one of the advantages of using page-locked host memory is the ability to map this type of memory to the device's address space which eliminates the need for copying data stored in this memory between the host and the device. Specifically, a block of page-locked host memory can be mapped to the device's address space by passing either the `cudaHostAllocMapped` flag to `cudaHostAlloc()` or, the `cudaHostRegisterMapped` flag to `cudaHostRegister()` [2].
Then, the block will have an address in host memory - given by `cudaHostAlloc()` (or `malloc()`) - and another in device memory. The address in device memory is accessible by using the function `cudaHostGetDevicePointer()`, which will provide a pointer that can be used in kernels. Even though this specific type of host memory is accessible from within a kernel, it is not stored on the device and, thus, accessing it is not as fast as accessing device memory.
Another advantage of using mapped memory is that copying the block of data between device memory and mapped host memory is done implicitly by kernels when needed, thus, the need to allocate a block in device memory is removed. However, a disadvantage of this characteristic is the potential read-after-write, write-after-read, or write-after-write complications that can arise when asynchronous concurrent execution (detailed in Section 1.2.4) is used [2]. These problems can be avoided by forcing memory



(a) In the upper image, two threads swap orders and thus do not access successive floats in global memory. However, according to *CUDA C++ Programming Guide* [2], this does not result in non-coalesced memory access. All other requirements are fulfilled. The lower image shows a scenario where the starting address is misaligned as the size of the memory segment is 64 bytes, but, the starting address is 132 which is not divisible by 64: $132/64 = 2.0625$. Therefore, access is misaligned and the memory segment is read in two sequential memory transactions: starting addresses 128 - 188 and starting addresses 192-252.



(b) In the upper image, threads `t2` - `t15` are misaligned with the original starting address. Thus, similarly to the lower image in Figure 1.12a, the memory transaction will be split into two sequential transactions. The lower image shows a situation where global memory houses 12-byte `struct` types. However, as Table 1.4 and Listing 1.1 show, 12-byte alignment is not supported, therefore global memory access is non-coalesced.

Figure 1.12: Examples of coalesced and non-coalesced access to global memory composed of 4-byte words (for example, `float`) with one exception being 12-byte structures. Taken from Martínez Manuel Ujaldón's *CUDA Optimizations, Debugging and Profiling* [25].

synchronizations, however, this can have an impact on performance.
Further caveats associated with the use of mapped memory can be found in *CUDA C++ Programming Guide* [2].

An overview of how the structure of hardware on a GPU and CUDA can be seen in Figure 1.13.

1.2.4 Asynchronous Concurrent Execution

Within CUDA multiple operations can be executed at the same time (concurrently) - namely [2]:

- Computation on the host

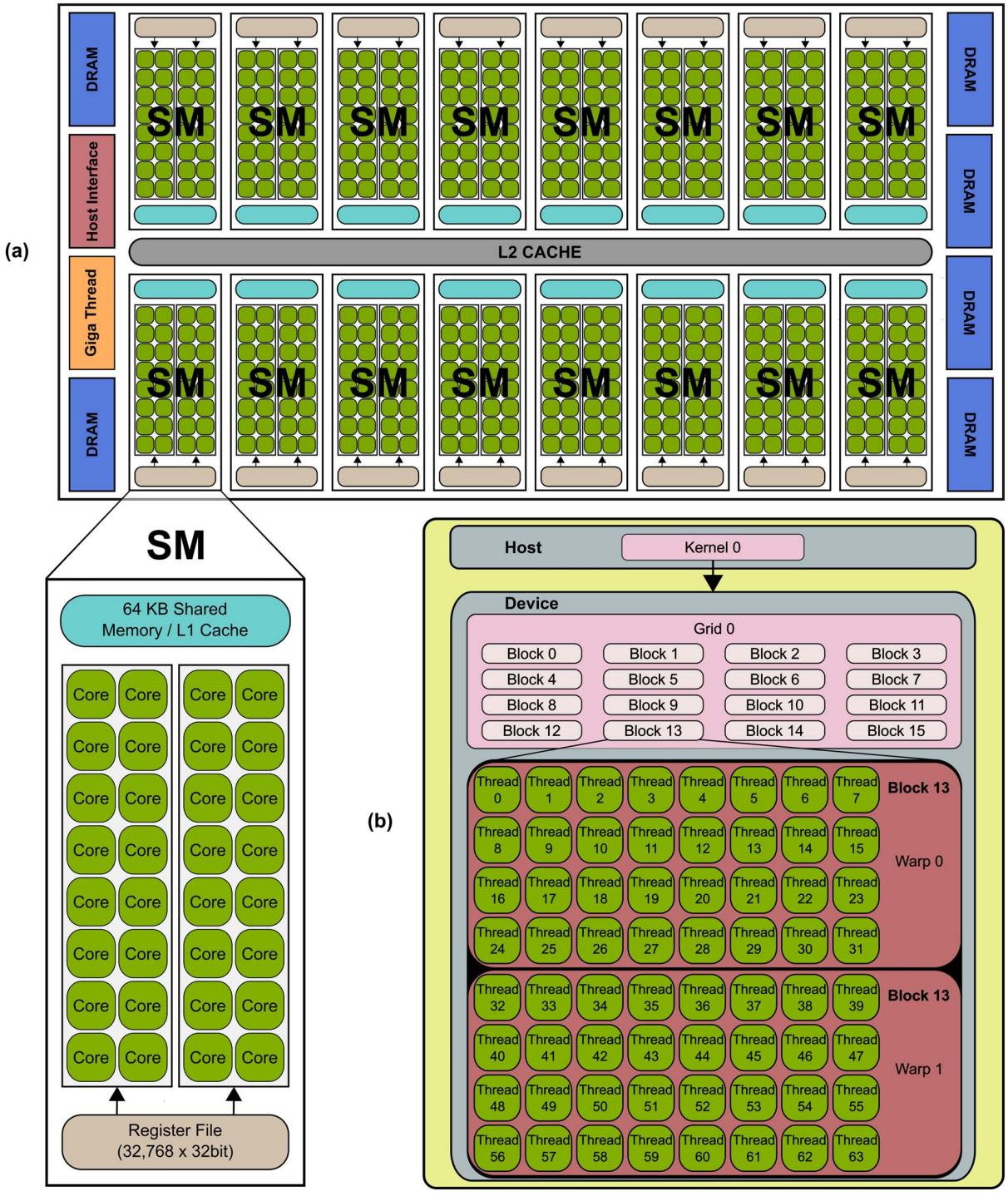


Figure 1.13: a) Usual architecture of an Nvidia Fermi GPU that comprises of SMs. Furthermore, each SM is made up of SP cores (Stream Processor cores) b) CUDA programming model controls the hardware of the GPU. Taken from *Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs* [26].

- Computation on the device

- Memory transfers from the host to the device and vice versa
- Memory transfers within the memory of a given device

However, it is important to note that there is a limit to concurrent load. According to *CUDA C++ Programming Guide* [2], the so-called *level of concurrency* is determined by the compute capability version of the device and its available feature set as detailed below.
This section will describe concurrent execution from different perspectives that are used in this project. Firstly, concurrent execution between the host and the device will be presented. Then, concurrent kernel execution will be introduced in greater detail and finally, the topic of streams will be described.

Concurrent execution between host and device Concurrent host execution means that once the host launches a kernel on the device, then it does not have to wait for the kernel to complete before moving on to the next instruction. CUDA has asynchronous library functions that make the host a controller which essentially sends computation jobs to the device. In other words, once a kernel is launched on the device, the host reclaims control immediately without waiting for the kernel to finish. This means that the host can go on to execute the next line of code.
Due to the asynchronous nature, this functionality is especially useful when it comes to queuing jobs for the device. The jobs are executed on the device by CUDA as soon as sufficient resources are available. Subsequently, the responsibility of managing the device is lifted from the host - allowing it to perform other tasks [2].
Host asynchrony is available for the following device operations [2]:

- Kernel launches
- Memory copies (only page-locked host memory):
  - Within a single device's memory
  - From host to device - memory block with a limited size of 64 KB
  - Performed by functions suffixed with `Async`
- Memory set function calls

Asynchronous concurrent execution between host and device can be disabled on a system for all CUDA applications (kernel launches) by using the `CUDA_LAUNCH_BLOCKING` environment variable (set to `1`). However, Nvidia does not recommend using this feature in production - only during debugging.
Another noteworthy aspect is that kernel launches are implicitly synchronous when running a CUDA application via a profiler, for example, the Nvidia Visual Profiler. This behavior can be overridden in the profiler by explicitly enabling concurrent kernel profiling [2].

Concurrent kernel execution Similarly to the concept where the device can be running a kernel while the host is performing another task, kernels can be executed simultaneously on the device. Not all Nvidia devices support this functionality (only some devices with compute capability 2 and higher), therefore, it must be checked per device using the `asyncEngineCount` device property (`0` if the device supports it). There are 3 main limitations when it comes to concurrent kernel execution: CUDA-defined maximum number of resident grids per device, no concurrent kernels from different CUDA contexts, and resources

available on the device. First, the CUDA-defined maximum resident grids per device will be explained. Since each kernel is launched on an individual grid of blocks of threads, then the maximum number of resident grids per device is effectively the maximum number of kernels that can be run concurrently on the device. The limit is a hard-set constant that depends on the compute capability version - Table 1.5.

Compute Capability	3.5 - 5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5 - 8.7
Max. resident grids per device	32	16	128	32	16	128	16	128

Table 1.5: Maximum number of resident grids per device depending on the CUDA compute capability version. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

The second limitation - CUDA context - signifies that if two CUDA applications are running at the same time on a system, then two kernels - one from each context - will not be run concurrently. Instead, the kernel that was launched second will be queued to execute until the first kernel has finished.
The last limitation - device resources - is not as severe as the previous two since kernels can still be launched concurrently, even if their cumulative required resources exceed the device's available resources. This is due to the fact that if more than one kernel is set to be launched concurrently, they are only run simultaneously if the available resources are sufficient for them all. If the resources available are not sufficient, then the kernels which would not get their requested resources are put aside until the resources are free. In this instance, resources represent memory and threads.
In terms of memory, an example of kernels that will often not run concurrently with other kernels are ones that require large amounts of local memory [2]. In terms of threads, the limiting factor is the maximum number of active threads - described at the end of the *Grid* paragraph in Section 1.2.2. For example, if two kernels are to be run concurrently, the total number of threads required by them at once must not exceed the maximum number of active threads on the device.
Therefore, whether kernels can be run concurrently is highly dependent on the resource requirements of each kernel and it is up to the device to manage all of these operations.

Streams

Streams can be used to effectively create and manage concurrency on the device, in other words, they are the means by which the above-mentioned operations are controlled. According to *CUDA C/C++ Streams and Concurrency* by Steve Rennich and Nvidia, a CUDA *stream* can be defined as "*A sequence of operations that execute in issue-order on the GPU*" [27]. To put it another way, a single stream can be thought as an open door through which instructions can be sent to the device - kernels.
While it is possible to have multiple streams active at once, the device has limited resources available (active threads and memory). Therefore, multiple streams being active at once does not necessarily mean multiple kernels running simultaneously. Nevertheless, if multiple streams are active and each is given a different kernel, then these kernels can be executed concurrently if the device's limits are not overstepped - detailed above in paragraph *Concurrent kernel execution* in Section 1.2.4.
Unlike individual kernels, global memory is used by all streams without division, i.e. all streams have access to the same global memory and no part of global memory belongs to a particular stream.
According to Nvidia's *CUDA C++ Programming Guide* [2], individual streams are not dependent on each other, meaning that instructions can be executed on them separately or concurrently. However, this behavior does not have to be consistent, for example, communication between kernels is not defined. For this reason, Nvidia themselves recommend not relying on the accuracy of CUDA applications when

attempting to make streams interact.
On the other hand, CUDA provides tools that can assist when the synchronicity of streams is required. This section will first present basic information on stream creation and destruction. Then, the topic of the default CUDA stream will follow and, finally, explicit and implicit synchronization will be detailed.

Creation and destruction A CUDA stream is created by initializing a stream object - type `cudaStream_t` in code - and then creating the stream itself using `cudaStreamCreate()`. The approach that is often used in examples by Nvidia is to create an array of streams as shown in Listing 1.2.

```
// Declare array of 2 stream objects
cudaStream_t streams[2];

// Create each stream
for( int i = 0; i < 2; ++i ) {
  cudaStreamCreate( &streams[i] );
}
```

Listing 1.2: Creation of streams. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

To make a kernel run on a specific stream, the stream address must be specified as one of the launch parameters of the kernel - detailed in Section 1.2.5. Launching a kernel on a specific stream is shown in Listing 1.3.

```
// Launch MyKernelA using the 0th stream using 1 block made up of 1 thread and 0 ↩
    bytes of dynamically allocated shared memory
MyKernelA<<< 1, 1, 0, stream[0] >>>( inputVariableA )

// Launch MyKernelB using the 1st stream
MyKernelB<<< 1, 1, 0, stream[1] >>>( inputVariableB )
```

Listing 1.3: Pseudo-code for launching two different kernels using two different streams. The instructions in this example would be executed from the host. Since each kernel is essentially an open door to the device for instructions, then, once `MyKernelA` is launched on `stream[0]`, the control is returned to the host without waiting for `MyKernelA` to finish. Subsequently, the host will immediately launch `MyKernelB` using `stream[1]`. In this example, each kernel is launched on a grid made up of one single-thread block with 0 bytes of dynamic shared memory allocated, thus, the device's resources will not be exhausted and both kernels will run concurrently. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

When streams are no longer needed, they are to be destroyed using `cudaStreamDestroy()` - shown below in Listing 1.4. If `cudaStreamDestroy()` is called while a stream is still performing tasks, then it will return without destroying the stream immediately - the destruction (release) of the stream's resources will be delayed until all work is completed on the stream.

```
// Destroy each stream
for( int i = 0; i < 2; ++i ) {
  cudaStreamDestroy( stream[i] );
}
```

Listing 1.4: Destruction of streams. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Default stream If a stream is not specified when launching a kernel, or when copying memory, then only the default stream - *Stream "0"* - is used and the instructions are executed in order with respect to the device (not concurrently as shown in the examples above) [2].
Nevertheless, it is possible to provide each host thread with its own default stream by setting the `--default-stream` compilation flag to `per-thread`, i.e. compiling using `nvcc ... --default-stream per-thread`.
The default option for the flag is `legacy`, which signifies that all host threads will use the special *NULL stream*. This stream is different from other streams as it uses implicit synchronization - detailed below in paragraph 1.2.4.

Explicit synchronization The first, and arguably the most used, type of synchronization when it comes to streams is explicit synchronization. In this instance, the "explicit" modifier signifies that the synchronization is issued using one of the following functions from code [2, 28]:

- `cudaDeviceSynchronize()` - Synchronization of the entire device. This function serves as a checkpoint in code where streams will wait until they all get to this point in code.
- `cudaStreamSynchronize( cudaStream_t stream )` - Synchronization of a particular stream. This function takes a single `stream` as its input parameter and serves as a checkpoint to wait for the completion of all actions called before it.
- `cudaStreamWaitEvent( cudaStream_t stream, cudaEvent_t event )` - Execution delayed for commands added to the input `stream` until the input `event` completes.

Additionally, CUDA provides a function to check whether all commands of a stream that precede the current line of code have finished: `cudaStreamQuery()`. Further functions that can be used to manage streams can be found in section *6.4 Stream Management* in the *CUDA Runtime API* [28].

Implicit synchronization The opposing type of synchronization is called implicit, and, as mentioned in paragraph *Default stream* above, it is used by default under certain conditions - unless explicitly overridden by one of the functions above. Specifically, Nvidia states in *CUDA C++ Programming Guide* [2] that if there are two streams, then two commands - each from one stream - cannot be run simultaneously if the host thread issues any of the following operations:

- Page-locked host memory allocation
- Device memory allocation
- Device memory set
- Memory copy between two addresses to the same device memory
- Any CUDA command to the NULL stream
- Switch between the shared memory configurations

The hypothetical reasoning behind this is that if any of the operations above would be performed in parallel, it could create irresolvable conflicts. For example, if each stream tried to allocate memory on

the device at the same location concurrently - impossible to be done in parallel without a conflict check. However, it can be argued that a conflict check is an unnecessary functionality as it is already present in some form when allocation happens sequentially.
Furthermore, there are certain operations that require a dependency check [2]:

- any other commands within the same stream as the launch being checked;
- any call to `cudaStreamQuery()` on that stream.

Thus, to improve application performance, Nvidia recommends developers issue all independent operations before dependent operations and delay any synchronization until necessary.

1.2.5 C++ CUDA Extensions

As previously mentioned at the beginning of Section 1.2, CUDA supports a variety of programming languages. Among them is C++, a widely-used, high-performance C-based language that implicitly allows low-level memory manipulation. CUDA provides many different extensions to C++, however, for this project only the core basics that were used during its development along with some other important extensions will be detailed.
This section is divided into two main categories: outer-kernel and inner-kernel extensions. The former is made up of memory-related operations (allocating, copying, and freeing data), kernel launch configurations, function extensions, streams, etc. - i.e. extensions not used within kernels. The latter consists of any operators, structures, and declarations that are used in kernels.

Outer-kernel Extensions

This category will first present a selection of important memory managing extensions along with those that were used during the development of this project. Then, kernel-specific extensions will be detailed. Finally, the last part will present function modifiers that state where a function can be executed. The extensions to C++ for streams were described above in the *Streams* part of Section 1.2.4.

Memory managing extensions CUDA offers many memory managing extensions to C++, however, there are a select few that are widely used for allocating, copying, and freeing data [2, 28, 1]:

- `cudaMalloc( void** devPtr, size_t size )` - Function that allocates `size` bytes in device memory and stores the address in the `devPtr` pointer.
  Note that since the pointer is to an address located in device memory, it is inaccessible from the host - access from the host would first require for the data to be copied using `cudaMemcpy()`.
  This function is widely used when it comes to allocating anything from single variables to large arrays.
  If the data was allocated successfully, then `cudaSuccess` is returned, otherwise one of `cudaErrorInvalidValue`, `cudaErrorMemoryAllocation` is returned - depending on the type of failure.
- `cudaMemcpy( void* dst, const void* src, size_t count, cudaMemcpyKind kind )` - Function that copies `count` bytes from the source memory address

(src) to the destination memory address (dst). The kind parameter specifies the direction of copying; it has to be one of the following: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Similarly to cudaMalloc(), if the data was copied successfully, then cudaSuccess is returned, otherwise one of cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection is returned - depending on the type of failure.

- cudaFree(void* devPtr) - Function that frees memory on the device that is pointed to by devPtr.
  If the memory was successfully freed, then cudaSuccess is returned, otherwise, cudaErrorInvalidValue is returned.
  It is also worth noting that in order to be freed using cudaFree() the memory devPtr is pointing to must have been allocated by one of CUDA's memory allocation APIs, for example, cudaMalloc(), cudaMallocAsync(), etc. - the complete list can be found in Nvidia's *CUDA Runtime API: API Reference Manual* [28].

- cudaMallocHost(void** ptr, size_t size) - Function that allocates size bytes of page-locked memory on the host and stores the address in the ptr pointer. The benefits and caveats of using page-locked memory are detailed in paragraph *Page-locked host memory* in Section 1.2.3.

Expanding on the memory operations mentioned above, CUDA provides memory space specifiers that can be used to denote in what memory a variable ought to be stored. Among such often-used specifiers are [2]:

- __device__ - Memory space specifier declaring that a variable is stored in global memory on the device. Since it resides in global memory, it is accessible by all threads from the grid and it is present for the lifetime of the CUDA application. It is noteworthy that if multiple CUDA devices are present in the system, then a variable with this specifier is a unique object for each device.

- __shared__ - Memory space specifier declaring that a variable is stored in the shared memory of a thread block. Since it resides in shared memory, it is only accessible by threads of a block as each block has its unique object of this variable.

Listing 1.5 shows an example of how memory managing functions can be used in a host-device code that copies arrays.

```
1  int main()
2  {
3    float *a_h, *b_h; // data that will be allocated on host
4    float *a_d, *b_d; // data that will be allocated on device
5    int N = 14, nBytes, i;
6
7    nBytes = N * sizeof( float ); // required allocation size
8    a_h = (float *) malloc( nBytes ); // allocating host data
9    b_h = (float *) malloc( nBytes );
10   cudaMalloc( (void **) &a_d, nBytes ); // allocating device data
11   cudaMalloc( (void **) &b_d, nBytes );
12
13   // filling up host data
14   for( i = 0, i < N; ++i ) {
15     a_h[i] = 100.f + i;
```

```
16    }
17
18    // copying data from host -> device -> device -> host
19    cudaMemcpy( a_d, a_h, nBytes, cudaMemcpyHostToDevice );
20    cudaMemcpy( b_d, a_d, nBytes, cudaMemcpyDeviceToDevice );
21    cudaMemcpy( b_h, b_d, nBytes, cudaMemcpyDeviceToHost );
22
23    // checking that all data is equal
24    for( i = 0; i < N; ++i ) {
25      assert( a_h[i] == b_h[i] );
26    }
27
28    free( a_h ); free( b_h ); // freeing data on host
29    cudaFree( a_d ); cudaFree( b_d ); // freeing data on device
30    return 0;
31  }
```

Listing 1.5: Example of code that utilizes CUDA memory managing extensions of C++. Taken from *Formats for storage of sparse matrices on GPU* [1] and Nvidia's *Getting Started with CUDA* presentation [17].

Kernels While the term *kernel* was introduced earlier in Section 1.2.1, the technical details will be covered in this part. In order to differentiate a kernel from a function, the `__global__` modifier must be used. Another distinction that kernels have compared to regular functions is the kernel launch configuration which has the following specific syntax: `<<< numBlocks, threadsPerBlock, sharedMemSize, stream >>>` where [2, 1]:

- `numBlocks` specifies the number of blocks in the grid and their structure;
- `threadsPerBlock` specifies the number of threads per block and their structure;
- `sharedMemSize` specifies the number of bytes that are to be dynamically allocated in shared memory for each block (on top of any statically allocated shared memory; this argument is not mandatory and its default value is 0);
- `stream` specifies the stream to which the kernel should be sent for execution. This argument is also not mandatory and defaults to 0 - the default stream.

The `sharedMemSize` parameter is of type `size_t`, `stream` is of type `cudaStream_t`, and parameters `numBlocks` and `threadsPerBlock` can be either of type `int` or `dim3`. The `dim3` type is a 3-dimensional `unsigned int` vector used to specify dimensions. It can be defined with up to 3 variables (1 for each dimension) with an implicit dimension value of 1, i.e., if a vector component (dimension) is not specified when declaring a `dim3` variable, it is by default initialized to 1.
For example, if `numBlocks` is of type `int`, then the grid is one-dimensional and composed of `numBlocks` blocks. Concordantly, if `threadsPerBlock` is type `int`, then all blocks in the grid will be one-dimensional with `threadsPerBlock` threads.
On the other hand, if `threadsPerBlock` is a `dim3` variable, then the thread structure of all blocks can be up to three-dimensional. An example of a kernel being called with the properties above can be seen in Listing 1.6.

```
// Kernel definition
__global__ void MyKernel( float a, float b, float c )
{
  // Kernel code here
}

int main()
{
  ...
  // Kernel invocation on a grid with 1 block of 8 * 1 * 1 threads
  int numBlocks = 1;
  dim3 threadsPerBlock( 8 ); // Defaults to (8, 1, 1)
  MyKernel<<< numBlocks, threadsPerBlock >>>( a, b, c );
  ...
}
```

Listing 1.6: Example of C++ pseudocode of a Kernel launch on a grid consisting of 1 one-dimensional block that is made up of 8 threads. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Function extensions In the part above, the `__global__` modifier was introduced as a specifier for functions that are to be considered kernels. Along with it, CUDA brings other function modifiers, or, as Nvidia calls them: *execution space specifiers* [2]:

- `__global__` - Modifier declaring that a function can be called from both the host and the device (as of compute capability 3.2 or higher). However, it can only be executed on the device. Additionally, a function denoted with this modifier must not belong to any class and it can only have a void return type. Furthermore, the launch configuration mentioned above in the *Kernels* paragraph must be specified for each call.
- `__device__` - Modifier declaring that a function can only be called from and executed on the device. This means that it can only be called from within kernels.
- `__host__` - Modifier declaring that a function can only be called from and executed on the host. This means that it can only be called from regular functions, not kernels.

If no modifier is included for a function, it is compiled for the host only. However, if both `__device__` and `__host__` are specified for a function, then it is compiled both for the host and for the device. The complete list of modifiers can be found in *CUDA C++ Programming Guide* [2].

Inter-kernel Extensions

This category consists of variables, structures operations, etc. that are most often used within a kernel. First, the means by which developers can identify individual threads will be detailed. Then, different memory-type identifiers will be briefly presented along with established operations used to avoid erroneous behaviors in kernels.

Thread identification Once a kernel is launched on the device, it can be run concurrently on thousands of threads. This presents a scenario where the developer must distinguish what each thread will perform within the kernel. For this purpose, the following variables - among others - are available within every kernel for each thread [2]:

- `threadIdx` - ID of a thread within a block stored in a 3-component vector (1 component for each of 3 dimensions). For example, if the thread block is 2-dimensional, then `threadIdx.x` stores the thread's ID in the first dimension and `threadIdx.y` stores the thread's ID in the second dimension - the pair of IDs is unique only within the thread's block.
- `blockIdx` - ID of a block within a grid stored in a 3-component vector (1 component for each of 3 dimensions). Similarly to `threadIdx`, the block's ID in the first dimension is retrieved using `blockIdx.x`, the second using `blockIdx.y`, and the third using `blockIdx.z`. Within a kernel, the block ID refers to the block of threads that the executing thread belongs to.
- `blockDimx` - Dimensions of a thread's block stored in a 3-component vector (1 component for each of 3 dimensions). For example, for a block made up of `32x16x2` threads: `blockDimx.x` = `32`, `blockDimx.y` = `16` and `blockDimx.z` = `2`.

The variables above can be combined to calculate the global ID of a thread, i.e. the thread's ID within a grid:

`globalID` = `blockIdx.x` * `blockDimx.x` + `threadIdx.x`

The `globalID` is useful when working with matrices, for example, adding two matrices. In such an example, each thread would take two elements (1 from each matrix), add them and then store the result into a new matrix - as shown in Listing 1.7.

```
// Kernel definition
__global__ void MatAdd( float A[N][N], float B[N][N], float C[N][N] )
{
  // Get thread ID for each dimension -> use as matrix indices
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  // Check if the IDs are within the dimensions of the matrices
  if( i < N && j < N ) {
    C[i][j] = A[i][j] + B[i][j];
  }
}

int main()
{
  ...
  // Number of threads per each 2-dimensional block is 16x16 = 256
  dim3 threadsPerBlock( 16, 16 );

  // Number of blocks in the grid depends on the matrix dimension (NxN)
  dim3 numBlocks( N / threadsPerBlock.x, N / threadsPerBlock.y );

  // Kernel launch: A + B = C
  MatAdd<<< numBlocks, threadsPerBlock >>>( A, B, C );
  ...
}
```

Listing 1.7: Example of C++ pseudocode of a kernel that adds two matrices using 2-dimensional thread blocks. Since each thread has a unique `globalID` for each of two dimensions, then those IDs can be used as indices for adding matrix elements. This simple example does not take into account the allocation and copying of data from host to the device. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Accompanying the code in Listing 1.7, Figure 1.14 visualizes how elements of a matrix can be divided into a grid comprising 2-dimensional thread blocks.

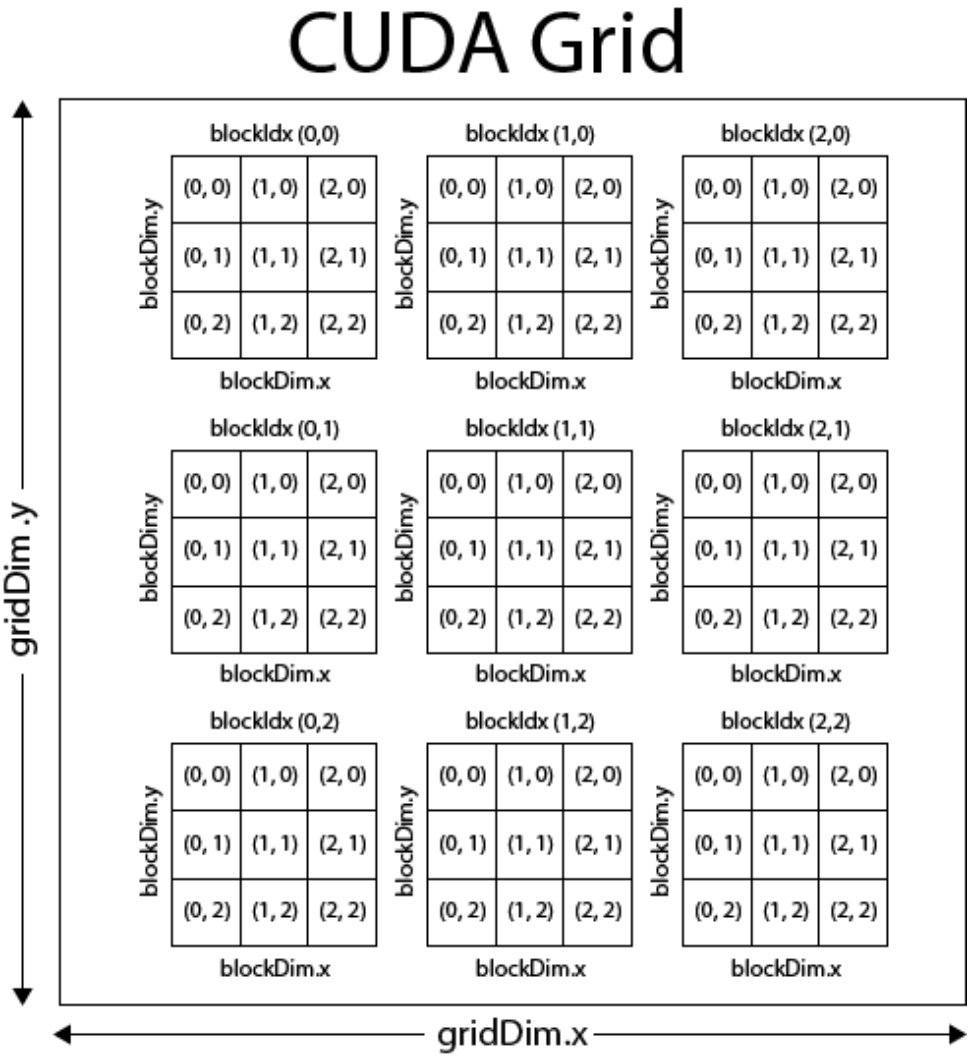


Figure 1.14: Grid made up of 2-dimensional thread blocks. Each square represents a thread with its IDs within its block (`threadIdx.x`, `threadIdx.y`). It can be imagined that each block is a submatrix and together all submatrices make up the full matrix. Taken from *CUDA Parallel Thread Management* [29].

Finally, as mentioned in paragraph *Global memory* in Section 1.2.3, the coalesced access of threads effectively means that neighboring threads access global memory. To add to this, it is important to mention that threads are considered neighboring in the first dimension. In other words, neighbors according to `threadIdx.x` values.

Memory specifiers and other operations Some memory space specifiers described above in paragraph *Memory managing extensions* in Section 1.2.5 can be used within kernels, specifically, the `__shared__` specifier. Its use within kernels is mostly related to statically allocated shared memory where the size of the variable or array is known during compilation [2]. An example showing this use of the specifier can be seen in Listing 1.11.
In paragraph *Shared memory* in Section 1.2.3, it was mentioned that threads of a block can share data purely among each other. However, this advantage can also bring problems, such as some threads overwriting data that other threads have not finished using, or, some threads reading data that other threads have not yet written - known as a *race condition* [30]. CUDA addresses this issue by enabling the synchronization of threads in a block which can be done by calling the built-in function: `__syncthreads()`. This function can be seen as a meeting checkpoint for all threads of a particular block, i.e., threads of a block will wait at this point until every single thread in their block has finished its work until this point. To put it more clearly: threads of a block are halted on the line in code that contains `__syncthreads()` until all threads of the block arrive at it [2].

1.2.6 Matrix Multiplication

In order to consolidate how CUDA can be used to accelerate the execution of a simple task, such as matrix multiplication, this section will present solutions to this task from Nvidia's *CUDA C++ Programming Guide* [2]. Furthermore, to showcase and stress the importance of abiding by Nvidia's recommendations when it comes to best practices and optimal performance, two examples will be presented. The first example will only use global memory to read the input matrix and write the resulting matrix, i.e. without shared memory. On the other hand, the second example will use shared memory to minimize accessing global memory.
It is important to note that neither example is fully functional as the logic of some functions has been omitted due to unnecessary complexity and irrelevance to showcasing the capabilities of CUDA. In other words, the code shown will not compile, nor will it run - the full code can be found in Nvidia's *CUDA C++ Programming Guide* [2]. The full working example (`matrixMul.cu`) encompassed by the build automation tool `make` can be unpacked during the installation of a CUDA toolkit.
Before introducing the specifics of each example, their similarities will be presented.
First, the task that will be performed is matrix multiplication:

$$\mathbb{C} = \mathbb{A} \cdot \mathbb{B} \,. \tag{1.1}$$

In the context of this operation, all matrix dimensions are assumed such that it is a legal operation. Additionally, for simplicity, the dimensions of the matrices are assumed to be multiples of the thread block size. This limitation can be avoided by allocating an extra row and column of blocks in the grid and setting a boundary condition in the kernel - each of the two examples (global vs shared memory) require slightly different approaches which will be described later.
Let each matrix be represented by a structure `Matrix`:

```
// Matrices are stored in row-major order:
// M( row, col ) = *( M.values + row * M.width + col )
typedef struct {
  int width;
  int height;
  float* values;
} Matrix;
```

Listing 1.8: Definition of the structure that will represent a matrix. The `width` variable stores the number of columns and `height` stores the number of rows the matrix has. The elements of the matrix are stored in row-major order in the single-precision (float) array: `values`. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Furthermore, let there be a host function `MatMul` that will:

1. Take matrices $\mathbb{A}$, $\mathbb{B}$, and $\mathbb{C}$ from Equation 1.1 as input structures: `A`, `B`, `C`.
2. Allocate memory for the matrix structures on the device.
3. Copy the matrix structures from host to device memory.
4. Invoke the matrix multiplication kernel with a constant thread block size set at compile time.
5. Copy the resulting matrix from device memory to host memory.
6. Free previously allocated memory from the device.

Listing 1.9 shows the implementation of the points mentioned above.

```
// Thread block size - 16x16 = 256 threads
#define BLOCK_SIZE 16

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul( const Matrix A, const Matrix B, Matrix C )
{
  // Allocate and copy A to device memory
  Matrix d_A;
  d_A.width = A.width; d_A.height = A.height;
  size_t size = A.width * A.height * sizeof(float);
  cudaMalloc( &d_A.values, size );
  cudaMemcpy( d_A.values, A.values, size, cudaMemcpyHostToDevice );

  // Allocate and copy B to device memory
  Matrix d_B;
  d_B.width = B.width; d_B.height = B.height;
  size = B.width * B.height * sizeof(float);
  cudaMalloc( &d_B.values, size );
  cudaMemcpy( d_B.values, B.values, size, cudaMemcpyHostToDevice );

  // Allocate C in device memory
  Matrix d_C;
  d_C.width = C.width; d_C.height = C.height;
  size = C.width * C.height * sizeof(float);
  cudaMalloc( &d_C.values, size );

  // Invoke kernel
  dim3 threadPerBlock( BLOCK_SIZE, BLOCK_SIZE );
  dim3 numBlocks( B.width / threadPerBlock.x, A.height / threadPerBlock.y );
  MatMulKernel<<< numBlocks, threadPerBlock >>>( d_A, d_B, d_C );

  // Read C from device memory
```

```
34   cudaMemcpy( C.values, d_C.values, size, cudaMemcpyDeviceToHost );
35
36   // Free device memory
37   cudaFree( d_A.values );
38   cudaFree( d_B.values );
39   cudaFree( d_C.values );
40 }
```

Listing 1.9: Definition of the function that will allocate and copy all matrices to the device, invoke the kernel and then free the device memory. The size of the thread block is constant and set during compile time using the `#define` macro. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

The `Matrix` structure and `MatMul` function conclude the equivalent part of the examples.

Example without Shared Memory

The first example uses only global memory access, i.e. without shared memory. This means that the kernel - invoked on line 31 in Listing 1.9 - receives the `Matrix` structures stored in global memory and each thread reads from and writes to the same global memory whenever they need to.
The kernel is launched on a grid of threads such that each thread is responsible for calculating exactly one element of the resulting `C` matrix. For example, let there be a thread with indices `row_id` and `col_id`. This thread will perform element-wise multiplication of row `row_id` from matrix `A` and column `col_id` from matrix `B`. Then, it will store the sum of these elements as the resulting value `C(row_id, col_id)` into matrix `C`.
The visualization of this approach is shown in Figure 1.15 and the kernel is presented in Listing 1.10.

```
1  // Matrix multiplication kernel called by MatMul()
2  __global__ void MatMulKernel( Matrix A, Matrix B, Matrix C )
3  {
4    // Each thread computes one element of C by accumulating results into Cvalue
5    float Cvalue = 0;
6    int row = blockIdx.y * blockDim.y + threadIdx.y;
7    int col = blockIdx.x * blockDim.x + threadIdx.x;
8    for( int i = 0; i < A.width; ++i ) {
9      Cvalue += A.values[row * A.width + i] * B.values[i * B.width + col];
10   }
11
12   C.values[row * C.width + col] = Cvalue;
13 }
```

Listing 1.10: Definition of the matrix multiplication kernel that uses global memory without shared memory. Each thread has a variable `Cvalue` stored in registers into which it calculates its specific `C(row, col)` matrix element. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

As can be seen in Listing 1.10, each thread performs `A.width + B.height` reads from global memory when calculating its `Cvalue`. Since CUDA threads are executed simultaneously in a warp of 32 threads, the memory access to global memory should coalesce for the threads within each warp. Nevertheless, this kernel is suboptimal as accessing global memory is considered suboptimal when avoidable.

At the end of the introduction to Section 1.2.6, it was mentioned that in order to eliminate the "matrix dimensions being multiples of `BLOCK_SIZE`" requirement, the kernel would need to include a boundary condition. In this example, the condition would be simply to allocate an extra column and row of thread

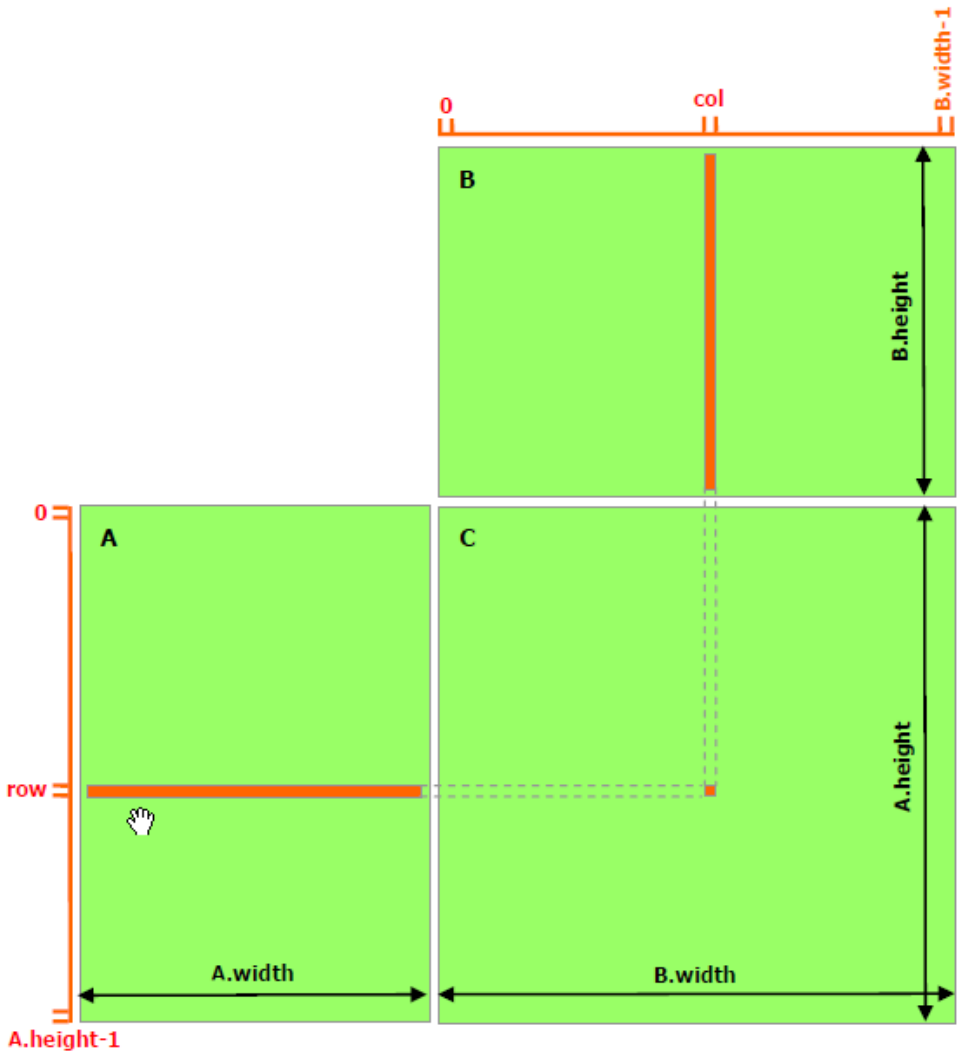


Figure 1.15: Visualization of the execution of the kernel that does not use shared memory - shown in Listing 1.10. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

blocks to the grid. This would mean that the last row and column of blocks would overlap the dimensions of the matrix. Then, the kernel would be terminated for threads that are outside the matrix dimensions, i.e. if either `row` is greater than `A.height` or `col` is greater than `B.width`. However, it is noteworthy that this will result in thread divergence, even though in this instance it will not have a large impact on overall performance as one of the execution paths would be a simple `return` statement.

Example with Shared Memory

The second example uses both global and shared memory. Similarly to the approach without using shared memory, each thread is responsible for calculating a single element of the resulting `C` matrix. However, in this example, each thread is also responsible for loading elements of matrices `A` and `B` from global memory into shared memory. Thus, shared memory is the primary means of obtaining elements

for calculations. Overall, the calculation is changed from each thread calculating its element to a block of threads iterating over submatrices of `A` and `B`.

In order to make use of shared memory, matrix `C` is divided into submatrices of `BLOCK_SIZE x BLOCK_SIZE` elements ($\mathbb{C}_{sub}$). Then, each submatrix $\mathbb{C}_{sub}$ is computed by a single thread block; each thread within a thread block is responsible for the computation of an element in $\mathbb{C}_{sub}$. In the previous approach, each thread performed element-wise multiplication of a row and a column to form a resulting element of matrix `C` - effectively a multiplication of a `1xn` and `nx1` matrix. The approach with shared memory expands this concept to blocks. In other words, the submatrix $\mathbb{C}_{sub}$ is a result of multiplying two rectangular submatrices of `A` and `B` (illustrated in Figure 1.16) - denoted $\mathbb{A}_{rect}$ and $\mathbb{B}_{rect}$ for this explanation:

1. $\mathbb{A}_{rect}$ is a `BLOCK_SIZE x A.width` submatrix of `A` that has the same row indices as $\mathbb{C}_{sub}$.
2. $\mathbb{B}_{rect}$ is a `B.height x BLOCK_SIZE` submatrix of `B` that has the same column indices as $\mathbb{C}_{sub}$.

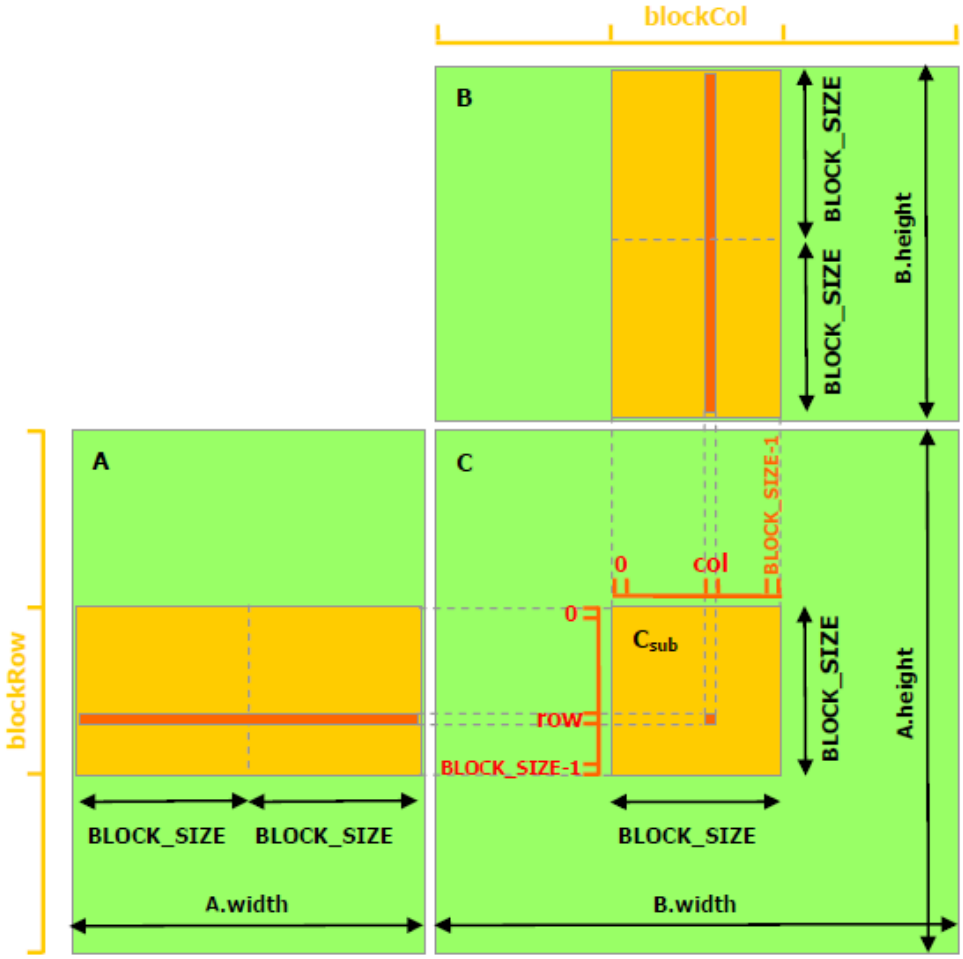


Figure 1.16: Visualization of the execution of the kernel that uses shared memory - shown in Listing 1.11. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

The rectangular submatrices $\mathbb{A}_{rect}$ and $\mathbb{B}_{rect}$ are split into `BLOCK_SIZE x BLOCK_SIZE` submatrices $\mathbb{A}_{sub}$ and $\mathbb{B}_{sub}$. A thread block will begin its computation by multiplying the leftmost $\mathbb{A}_{sub}$ with the uppermost $\mathbb{B}_{sub}$, then it will move to the next $\mathbb{A}_{sub}$ on the right and to the next B_{sub} below. After every iteration, each thread appends its partial sum to the `Cvalue` variable that it is responsible for in the $\mathbb{C}_{sub}$ submatrix. The computation of $\mathbb{C}_{sub}$ is finished once the thread block reaches and multiplies the rightmost $\mathbb{A}_{sub}$ and the lowermost $\mathbb{B}_{sub}$ submatrix.
The reasoning behind splitting $\mathbb{A}_{rect}$ and $\mathbb{B}_{rect}$ into multiple $\mathbb{A}_{sub}$ and $\mathbb{B}_{sub}$ respectively stems from the use of shared memory in every iteration for a particular block:

1. All threads of the block load $\mathbb{A}_{sub}$ and $\mathbb{B}_{sub}$ from global memory to a two-dimensional array residing in the block's shared memory.
2. Each thread of the block performs its computation, i.e. it multiplies row `row_id` from $\mathbb{A}_{sub}$ with column `col_id` from $\mathbb{B}_{sub}$ and sums the resulting values (`row_id` and `col_id` are the thread's IDs).
3. Each thread stores the temporary result to its local variable `Cvalue` stored in its registers.
4. If there still is an $\mathbb{A}_{sub}$ to the right and a $\mathbb{B}_{sub}$ below, then, all threads of the block move to them and continue to step 1.
5. Otherwise, the computation is finished and all threads store their local variable `Cvalue` to the element that they are responsible for in $\mathbb{C}_{sub}$ that resides in global memory.

The visualization of this approach is shown in Figure 1.16 and the kernel is presented in Listing 1.11.

```
1  // Matrix multiplication kernel called by MatMul()
2  __global__ void MatMulKernel( Matrix A, Matrix B, Matrix C )
3  {
4    // Block row and column - indices of the submatrices within A and B respectively
5    int blockRow = blockIdx.y;
6    int blockCol = blockIdx.x;
7
8    // Each thread block computes one submatrix Csub of C
9    Matrix Csub = GetSubMatrix( C, blockRow, blockCol );
10
11   // Each thread computes one element of Csub by accumulating results into Cvalue
12   float Cvalue = 0;
13
14   // Thread row and column within Csub
15   int row = threadIdx.y;
16   int col = threadIdx.x;
17
18   // Loop over all the submatrices of A and B that are required to compute Csub by↩
       multiplying each pair of submatrices together and accumulate the results
19   for( int i = 0; i < ( A.width / BLOCK_SIZE ); ++i ) {
20
21     // Get submatrix Asub of A
22     Matrix Asub = GetSubMatrix( A, blockRow, i );
23
24     // Get submatrix Bsub of B
25     Matrix Bsub = GetSubMatrix( B, i, blockCol );
26
27     // Shared memory used to store Asub and Bsub respectively
```

```
28      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
29      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
30
31      // Load Asub and Bsub from device memory to shared memory
32      // Each thread loads one element of each submatrix
33      As[row][col] = GetElement( Asub, row, col );
34      Bs[row][col] = GetElement( Bsub, row, col );
35
36      // Synchronize to make sure the submatrices are loaded before starting the ↩
            computation
37      __syncthreads();
38      // Multiply Asub and Bsub together
39      for( int k = 0; k < BLOCK_SIZE; ++k ) {
40        Cvalue += As[row][k] * Bs[k][col];
41      }
42
43      // Synchronize to make sure that the preceding computation is done before ↩
            loading two new submatrices of A and B in the next iteration
44      __syncthreads();
45    }
46
47    // Write Csub to device memory - each thread writes one element
48    SetElement( Csub, row, col, Cvalue );
49  }
```

Listing 1.11: Definition of the matrix multiplication kernel that uses both global memory with shared memory. The `GetSubMatrix(mtx, row, col)` is a function that returns a `BLOCK_SIZE x BLOCK_SIZE` submatrix of a matrix that is located `col` submatrices to the right and `row` submatrices down from the upper-left corner of the specified `Matrix`. The `GetElement(sub_mtx, row, col, val)` function returns the element found at a matrices `row` and `col` indices. Taken from Nvidia's *CUDA C++ Programming Guide* [2].

Unlike the previous example, the code shown in Listing 1.11 has fewer accesses to global memory which leads to a performance improvement as detailed later in the *Comparison of Both Examples* part of Section 1.2.6. Concretely, in the kernel above, each thread performs only `A.width/BLOCK_SIZE + B.height/BLOCK_SIZE` reads from global memory when loading submatrices $\mathbb{A}_{sub}$ and $\mathbb{B}_{sub}$ into shared memory. This means that all threads will perform `BLOCK_SIZE` times fewer accesses to global memory and `A.width + B.height` accesses to shared memory instead. Equivalently to the example before, access to global memory is coalesces. Thus, by Nvidia's recommendations, this kernel can be considered close to optimal.

In order to eliminate the "matrix dimensions being multiples of `BLOCK_SIZE`" requirement, the kernel would need to include a boundary condition. In the earlier approach, it was a matter of allocating extra blocks and terminating the kernel for threads that would reach out of matrix bounds. However, for this approach - with shared memory - that method would result in incorrect results as the threads that would reach out of bounds for one matrix read elements into shared memory from the other matrix. Thus, the boundary condition is split into:

1. If the thread would reach out of bounds of a particular matrix, load the edge element, i.e. the last element of the row/col before the thread would reach out of bounds. This condition would be used when the elements are being loaded from global into shared memory.

2. Then let the computation continue as normal - to avoid thread divergence of threads.

3. Once the computation is done, terminate the threads that would reach out of the bounds of matrix `C` before they write their results to global memory.

This solution ensures correct results and avoids a case of thread divergence in the main loop which would occur multiple times during the computation. However, the thread divergence will still occur when deciding which value to load from global memory and when terminating the kernel for some threads. Nevertheless, similarly to the previous approach, in this instance thread divergence will not have a large impact on overall performance as one of the execution paths would be a simple assignment operation in the first condition and a `return` statement in the second condition.

Comparison of Both Examples

In order to show the performance difference, benchmarks for both examples using single precision were run on a set of matrices with varying dimensions - from 160 by 160 to 16,000 by 16,000. The hardware and software specifications of the machine used for the benchmarks can be found in Table 1.6. The code used for the comparison was taken from Nvidia's GitHub repository *CUDA Samples*[1], specifically from `Samples/0_Introduction/matrixMul/`.

CPU	Ryzen 9 5900x @ 3.7 GHz (12 cores, 24 threads)
RAM	32GB RAM
GPU	Nvidia GeForce GTX 1070 8GB GDDR5 (256.3 GByte/s)
Operating System	Ubuntu 20.04.4 LTS (Focal Fossa)
Compiler	GCC 8.4.0
CUDA	CUDA 11.5

Table 1.6: Specifications of the platform that the matrix multiplication benchmarks were run on.

The operation measured during the benchmark was only the multiplication of matrices:

$$\mathbb{C} = \mathbb{A} \cdot \mathbb{B} \, .$$

In other words, allocation and copying of the matrices were not included in the measurement. Furthermore, the operation was looped 10 times - the measured FLOPS and run times were averaged from the 10 recorded loops. The values in Matrix $\mathbb{A}$ were all set to 1 and to 0.1 in matrix $\mathbb{B}$. The benchmark results in FLOPS can be found in Figure 1.17 and Table 1.7 shows the results in milliseconds.

As can be seen in Figure 1.17 and Table 1.7, the approach that utilizes shared memory achieved between 2-4 times the number of GFLOPs per second and similarly faster execution times compared to the approach that does not use shared memory and instead relies only on global memory. Additionally, the difference seems to be increasing with growing matrix dimensions, however, this statement has not been verified for dimensions greater than 16,000 by 16,000.

1.3 LU Decomposition

It can be argued that, in recent years, computational systems and software layers allowing developers to use them to their full potential have developed significantly - Nvidia GPUs and CUDA. Subsequently,

[1]CUDA Samples GitHub repository URL: `https://github.com/NVIDIA/cuda-samples`

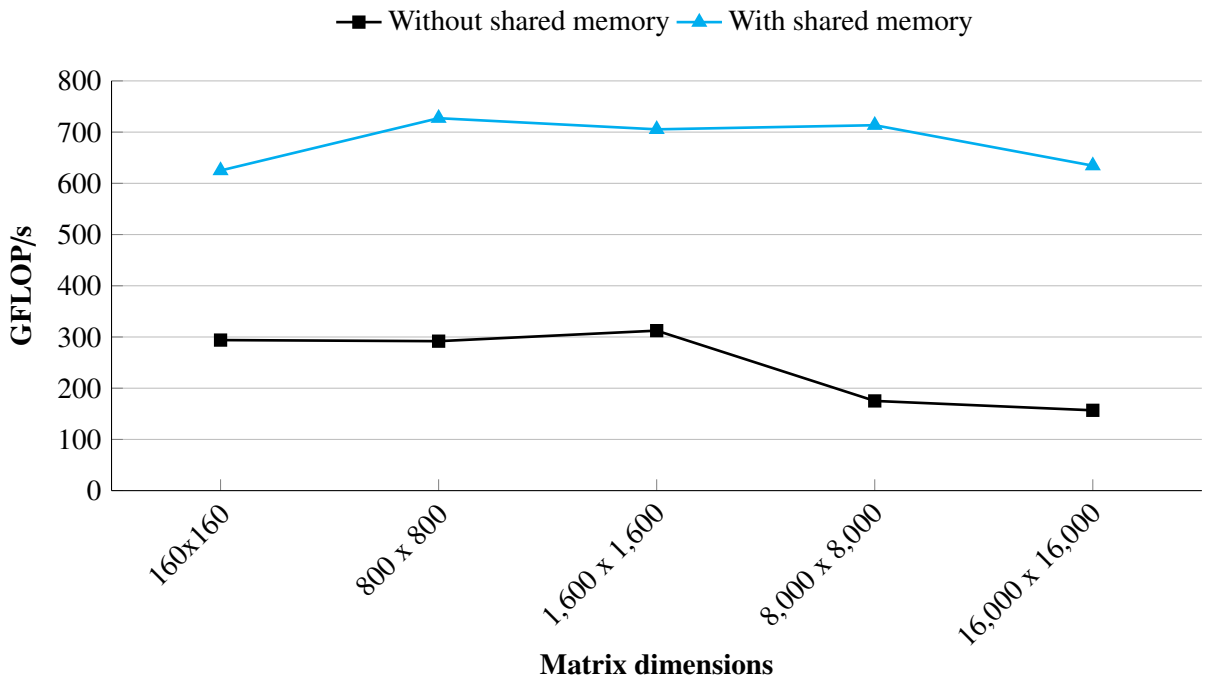


Figure 1.17: Matrix multiplication results of matrices with varying dimensions; matrices are filled with 1 element. The values - presented in GFlop/s - are averages from 10 loops of the operation.

Matrix dimensions	Without shared memory	With shared memory
160 x 160	0.028	0.013
800 x 800	3.509	1.408
1,600 x 1,600	26.224	11.613
8,000 x 8,000	5,846.247	1,435.221
16,000 x 16,000	52,226.867	12,910.967

Table 1.7: Matrix multiplication execution times (in milliseconds) of matrices with varying dimensions; matrices are filled with 1 element. The values are averages from 10 loops of the operation.

many novel uses have been found for such powerful computing systems, especially in areas that require results to be available quickly on demand. An example of such an area is solving systems of linear equations. Being one of the fundamental parts of numerical linear algebra, the requirement for their solution arises not only in computer science but also in physics, engineering, chemistry, etc.
While there are many different methods capable of solving a system comprising more than one linear equation, in this project, Lower-Upper (LU) decomposition will be detailed.
In order to show how LU decomposition can be used to solve a simple system of linear equations an example will be presented.

First, it is necessary to introduce the system. For the purpose of the explanation, let the coefficients of the following system of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1\,, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2\,, \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3\,, \end{aligned} \tag{1.2}$$

be rewritten into the matrix form

$$\mathbb{A}\mathbf{x} = \mathbf{b}\,, \tag{1.3}$$

where

$$\mathbb{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{1.4}$$

In order to be able to use LU decomposition, matrix $\mathbb{A}$ must be a square matrix ($n \times n$) that is also strongly regular. This requirement can be relaxed to only requiring regularity by properly ordering the rows and columns of the matrix using a permutation matrix $\mathbb{P}$, however, such a procedure will not be present in this project, and therefore all coefficient matrices are required to be strongly regular.

Second, using a *decomposition algorithm* the coefficient matrix $\mathbb{A}$ is decomposed into the product of a lower triangular matrix $\mathbb{L}$ and an upper triangular matrix $\mathbb{U}$:

$$\mathbb{A} = \mathbb{L}\mathbb{U}\,, \tag{1.5}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{1.6}$$

Then, substituting Equation 1.3 into Equation 1.3 yields:

$$\mathbb{L}\mathbb{U}\mathbf{x} = \mathbf{b}\,. \tag{1.7}$$

.

Third, the matrix form from Equation 1.7 is used to obtain the solution to the system of linear equations using forward and backward substitution:

1. Solve the equation $\mathbb{L}\mathbf{y} = \mathbf{b}$ (where only $\mathbf{y}$ is not known):

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{1.8}$$

2. Solve the equation $\mathbb{U}\mathbf{x} = \mathbf{y}$ (where only $\mathbf{x}$ is not known):

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}. \tag{1.9}$$

It is noteworthy that since the values on the right-hand side (vector $\mathbf{b}$) are only required in step 3 (equations 1.8 and 1.9), they are not required for the process of decomposition itself. Thus, if there is more than one right side in the system of linear equations, matrix $\mathbb{A}$ needs to be decomposed into matrices $\mathbb{L}$ and $\mathbb{U}$ only once; the same principle is valid even if the right side is not known ahead of time. In other words, $\mathbb{L}$ and $\mathbb{U}$ can be used to solve the system for different right-hand sides without the need for repeated decomposition. Additionally, this concept can be seen as an advantage for LU decomposition compared to Gaussian elimination as the latter requires the right-hand side to obtain the system's upper triangular matrix and subsequently use it for backward substitution. This concept is described, for example, by George Lindfield and John Penny in *Numerical Methods: Linear Equations and Eigensystems* [31].

In summary, from the steps above, it can be argued that the *decomposition algorithm* mentioned in step 2 is one of the key components of the procedure and as such it will be the main topic of this project.

1.3.1 Crout Method

This section aims to explain in greater detail the decomposition algorithm mentioned above in step 2. There are many different procedures to decompose matrix $\mathbb{A}$ into matrices $\mathbb{L}$ and $\mathbb{U}$ such that $\mathbb{A} = \mathbb{L}\mathbb{U}$. One such procedure, developed by Prescott Durand Croutis, is the *Crout method* - sometimes also referred to as *Crout matrix decomposition* or *Crout factorization* [32].
This method differs from similar procedures in that the resulting $\mathbb{U}$ matrix is a unit upper triangular matrix, while $\mathbb{L}$ remains to be a lower triangular matrix. A unit triangular matrix differs from a regular triangular matrix in that its main diagonal is comprised solely of ones. In other words, all elements on the main diagonal of $\mathbb{U}$ are equal to 1 ($u_{11} = u_{22} = u_{33} = 1$):

$$\begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.10}$$

Algorithm The Crout matrix decomposition algorithm is based on sequentially computing matrices $\mathbb{L}_{n\times n}$ and $\mathbb{U}_{n\times n}$ using elements from matrix $\mathbb{A}_{n\times n}$. At the core of this computation are the following formulas and their conditions for l_{ij} and u_{ij}:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \qquad i \geq j, \tag{1.11}$$

$$u_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right) \qquad i < j, \tag{1.12}$$

$$u_{ij} = 1 \qquad i = j, \tag{1.13}$$

$$i, j \in \widehat{n}.$$

Specifically, the algorithm first computes a column j in $\mathbb{L}$ and then row j in $\mathbb{U}$. This process repeats itself for j from 1 to n. The visualization of the algorithm's advance is shown in Figure 1.18 and the pseudocode implementing the algorithm can be seen in Listing 1.12.

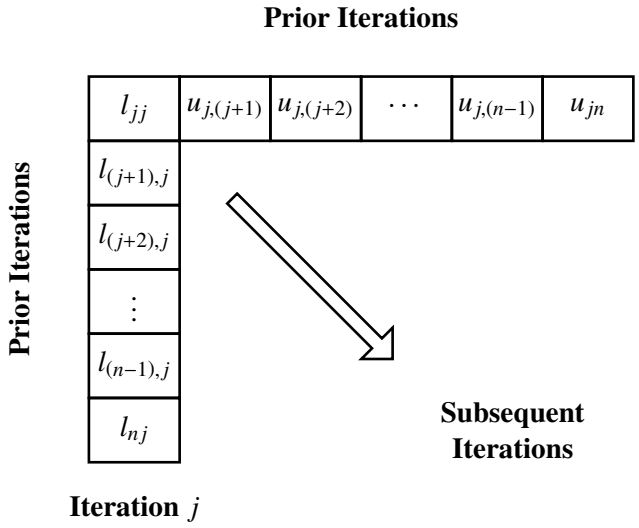


Figure 1.18: Sequence of computations of the Crout method. First, column j in $\mathbb{L}$ is computed and then row j of $\mathbb{U}$. Taken from Vismor's *Crout's LU Factorization* [33].

```
1  void crout_method( A, L, U )
2  {
3    int i, j, k;
4    double sum = 0;
5
6    // Fill main diagonal of U with 1s
7    for( i = 0; i < n; ++i ) {
8      U[i][i] = 1;
9    }
10
11   // Loop through the main diagonal
12   for( j = 0; j < n; ++j ) {
13
14     // Compute column j in L
15     for( i = j; i < n; ++i ) {
16       sum = 0;
17       for( k = 0; k < j; ++k ) {
18         sum += L[i][k] * U[k][j];
```

```
19        }
20
21        L[i][j] = A[i][j] - sum;
22      }
23
24      // Compute row j in U
25      for( i = j; i < n; ++i ) {
26        sum = 0;
27        for( k = 0; k < j; ++k ) {
28          sum = sum + L[j][k] * U[k][i];
29        }
30
31        if( L[j][j] == 0 ) {
32          printf( "det(L) close to 0!\n Can't divide by 0...\n" );
33          exit( EXIT_FAILURE );
34        }
35
36        U[j][i] = ( A[j][i] - sum ) / L[j][j];
37      }
38    }
39  }
```

Listing 1.12: C++ pseudocode implementing Crout's matrix decomposition algorithm. It assumes that `A[n][n]` is a two-dimensional array that represents the invertible square coefficient matrix $\mathbb{A}$; `L[n][n]` and `U[n][n]` are also two-dimensional arrays representing matrices $\mathbb{L}$ and $\mathbb{U}$ respectively. Furthermore, it is assumed that `L` and `U` are populated with zeros. Derived from *Crout's LU Factorization* [33], *Numerical recipes: the art of scientific computing* [32], and *Crout matrix decomposition* [34].

1.3.2 Numerical Method

When solving systems of linear equations there are two main groups of methods:

1. Direct methods - theoretically, the exact solution will be obtained after a finite amount of steps (e.g. Crout method).
2. Iterative methods - theoretically, these methods converge to the solution, however, the exact solution is not guaranteed to be found (numerical methods).

The Crout method as described in Section 1.3.1 is direct, meaning that it provides an exact solution. Furthermore, it is strictly sequential, meaning that there is seemingly no room for parallelization.
However, Hartwig Anzt et al. in their paper *ParILUT - A Parallel Threshold ILU for GPUs* [35] describe different approaches to generating incomplete factorizations. An example of such a group of approaches is referred to as *ParILU algorithms* which are distinct in the fact that they abandon the methodology akin to Gaussian elimination. Rather, these methods use "*fixed-point iterations to approximate the incomplete factors on a pre-defined sparsity pattern*" - iterative methods. The authors in the paper explain the methods belonging to this group as algorithms that do not take every nonzero into account.
However, for this paper, it was decided to derail from this principle - exclude any sparsity pattern conditions - and focus purely on the iterative aspect of the methodology. In other words, use the fixed-point iterating algorithm that - according to the authors - converges to a sufficiently approximate solution of $\mathbb{A} = \mathbb{LU}$ without taking into account sparsity patterns.
Specifically, the algorithm is as follows:

1. Provide an initial guess (estimate) of matrices $\mathbb{L}^0$ and $\mathbb{U}^0$. For example, using $\mathbb{A}$:

$$\begin{aligned} l_{ij}^0 &= a_{ij} && i < j\,, \\ u_{ij}^0 &= a_{ij} && i > j\,, \\ u_{ij}^0 &= 1 && i = j\,. \end{aligned}$$

2. Using the formulas in Equations 1.11 and 1.11 calculate the next iteration of $\mathbb{L}^t$ and $\mathbb{U}^t$ (where $t \in \widehat{\mathbb{N}}_0$ denotes the iteration): $\mathbb{L}^{t+1}$ and $\mathbb{U}^{t+1}$

$$\begin{aligned} l_{ij}^{t+1} &= a_{ij} - \sum_{k=1}^{j-1} l_{ik}^t u_{kj}^t && i \geq j\,, \\ u_{ij}^{t+1} &= \frac{1}{l_{ii}^t}\left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}^t u_{kj}^t\right) && i < j\,. \end{aligned}$$

3. If either $\left|\mathbb{L}^{t+1} - \mathbb{L}^t\right|$ or $\left|\mathbb{U}^{t+1} - \mathbb{U}^t\right|$ is greater than some tolerance, e.g. 0.001, then go back to step 2.

4. If both differences are smaller than some tolerance, then the algorithm has iteratively converged to an approximate solution of $\mathbb{A} = \mathbb{L}\mathbb{U}$.

On one hand, the number of iterations it takes the algorithm to converge to a solution is not possible to accurately predict and therefore could lead to poor performance if performed sequentially. On the other hand, since $\mathbb{L}^{t+1}$ and $\mathbb{U}^{t+1}$ are calculated independently, the operation can be run in parallel.
Another noteworthy, but unverified, aspect of this method's convergent nature is related to rounding errors. Specifically, since Crout's method is direct, it can be theorized that rounding errors may result in it providing less accurate results compared to its numerical modification. This thought stems from the fact that the numerical method converges and thus - under specific tolerance rules - may arrive at a more accurate solution. However, this is purely a hypothesis and remains to be verified.
Joining together the highly parallel nature of CUDA-enabled Nvidia GPUs and such a heavily parallelizable algorithm is the main focus of this project and will be detailed further in the following chapters.

Chapter 2

Implementation

This chapter will present the implementation of the *Decomposition* project that was built around the algorithms described in Sections 1.3.1 and 1.3.2. First, the *TNL* library[1] will be introduced briefly as it contains a structure that was paramount to the implementation of this project. Then, the description of the *Decomposition* project itself will follow. Finally, the last section will present how the iterative algorithm, mentioned in Section 1.3.2, was fine-tuned for optimal performance using CUDA.

2.1 TNL Library

The Template Numerical Library (TNL) is a collaborative, open-source project licensed under the MIT license. It aims to use modern programming paradigms combined with the computational power of parallel architectures to provide highly performant numerical solvers and simulations. In terms of parallel architectures, TNL supports both multi-core CPUs (using OpenMP[2]) and GPUs (as of July 2022 only CUDA-enabled Nvidia GPUs with compute capability 9.0 or higher [36]). As TNL is written in C++ (except for a few helping Python/Bash scripts) it is designed to benefit from C++ templates and the advantages they bring. For example, the use of template specializations allows for minimal overhead runtime which makes the execution of architecture-dependent code simple and fast [37]. Further up-to-date information regarding TNL can be found on the project's website[3].
While TNL contains, among others, many different data structures. For this project, only the one used during development will be introduced.

Dense matrix As mentioned at the end of Section 1.3.2, the goal of this project is to combine the parallel nature of CUDA-enabled Nvidia GPUs and the heavily parallelizable algorithm described in the same section (*Iterative Crout method*).
Therefore, to store all matrices involved in the computation, an efficient, robust, and intuitive data structure was required. While in terms of matrices TNL mainly revolves around sparse implementations (i.e. efficiently storing and performing operations with matrices that have more zero than nonzero elements), it also possesses the opposite: *dense matrices*. As the name suggests, dense matrices are matrix

[1] TNL library GitLab repository URL: `https://gitlab.com/tnl-project/tnl`
[2] OpenMP website URL: `https://www.openmp.org/`
[3] TNL project website URL: `https://tnl-project.org/`

structures densely populated with elements. In other words, a dense matrix stores every single element, regardless of its value (zeros and nonzeros are stored). In TNL, the class representing the dense matrix data structure is called `DenseMatrix` and it is located in the `TNL::Matrices` namespace. Similarly to the majority of classes representing matrices in TNL, it inherits from the `Matrix` class; the visualization of inheritance can be found in Figure 2.1.

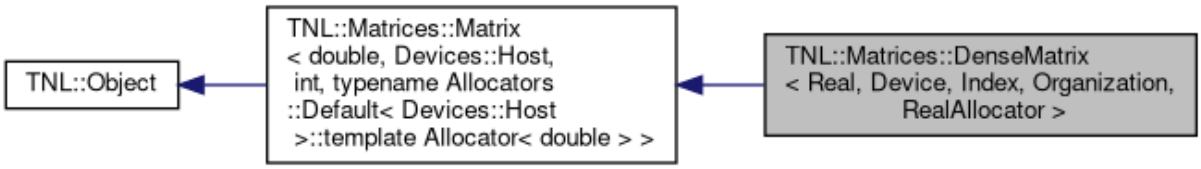


Figure 2.1: Inheritance diagram for the `TNL::Matrices::DenseMatrix` class. Taken from the *Template Numerical Library Documentation* [36].

In the diagram shown in Figure 2.1 the `DenseMatrix` class contains the following template parameters [36]:

- `Real` - Datatype of the matrix's elements, for example, `float` or `double` (default).
- `Device` - Type of device that the matrix structure is stored on. Its value can be either `TNL::Devices:Host` (default) or `TNL::Devices::Cuda`. The former indicates that the structure is stored in host memory, whereas the latter indicates that the data is stored in device memory. Furthermore, this parameter dictates which of the class's methods can be called and what device-dependent code should be executed.
- `Index` - Datatype of the matrix's index, for example, `short`, `int` (default), or `long`.
- `Organization` - Type of matrix element ordering. It can be either `RowMajorOrder` or `ColumnMajorOrder`. The former indicates that the elements of the matrix are stored in row-major order in a `DenseMatrix` instance's 1D array `values`, whereas the latter dictates that the elements are stored in the array in column-major order. Implicitly, row-major order is used when the structure is allocated on the host and column-major order if it is allocated on the device.
- `RealAllocator` - Allocator for the matrix elements.

The `DenseMatrix` class contains an assortment of methods with varying functionalities - from the initialization of an entire matrix via arrays to per-element lambda functions and CUDA-tailored algorithms. In terms of this project, the most used methods were those that set and get elements, which can be done by using `setElement()` and `getElement()` respectively. Alternatively, for setting and getting elements, `DenseMatrix` possesses the overloaded operator: `operator()`, which allows the developer to write cleaner code that conveys its purpose more clearly. From the perspective of performance, both approaches are congruous as they take the same arguments and compute the index of the element in the instance's 1D array. Nonetheless, there is a difference between the two approaches: `operator()` has the `__cuda_callable__` prefix, which specifies that it can be called from both host functions and CUDA kernels. However, this prefix comes with certain caveats:

- Data allocated on the host can only be accessed by `operator()` from host functions.

- Data allocated on the device can only be accessed by `operator()` from the device, specifically, from CUDA kernels. For example, if an instance of `DenseMatrix` has `Device` set to `TNL::Devices::Cuda` and `operator()` would be used to access data from a host function, then the program would fail.

The limitations above are not present for the other approach (`setElement()` and `getElement()`). In other words, both `setElement()` and `getElement()` can be used from host functions and CUDA kernels, irrespective of where the `DenseMatrix` instance is allocated. However, this convenience has a pitfall: accessing data stored on the other device type. For example, let an instance of `DenseMatrix` be stored in device memory. Then, if it is accessed from a host function using `getElement()`, the procedure will be slow and only the value of the element will be returned. Listing 2.1 shows an example of both approaches - including the caveats and pitfalls.

```
1  #include <iostream>
2  #include <TNL/Matrices/DenseMatrix.h>
3  #include <TNL/Devices/Host.h>
4
5  template< typename Device >
6  void setElements()
7  {
8    // Create an instance of 5x5 DenseMatrix
9    TNL::Matrices::DenseMatrix< double, Device > matrix( 5, 5 );
10
11   // Set elements of the main diagonal using setElement()
12   for( int i = 0; i < 5; ++i ) {
13     matrix.setElement( i, i, i );
14   }
15
16   // Increment elements of the main diagonal using operator()
17   for( int i = 0; i < 5; ++i ) {
18     matrix( i, i ) += i;
19   }
20 }
21
22 int main( int argc, char* argv[] )
23 {
24   // This example will run without any problems
25   std::cout << "Set elements on host:" << std::endl;
26   setElements< TNL::Devices::Host >();
27
28 #ifdef HAVE_CUDA
29   // This example will fail on the second setting of elements
30   std::cout << "Set elements on CUDA device:" << std::endl;
31   setElements< TNL::Devices::Cuda >();
32 #endif
33 }
```

Listing 2.1: Code example showing the use of `setElement()`, and `operator()`; `getElement()` and getting a matrix element using `operator()` would be done similarly. The example requires the TNL library to already be installed. As mentioned in the code, the `setElements()` function will run correctly when evoked using the `TNL::Devices::Host` template parameter. However, if the `DenseMatrix` instance is allocated on the device, then the first setting of elements (using `setElement()`) will be slow due to the data transfer between the host and the device - each call will mean a unique data transfer. Furthermore, the second setting of elements (using `operator()`) will fail as `operator()` will be called from a host function on a `DenseMatrix` instance allocated on the device. For the second setter to work, it would need to be called from within a kernel. Taken from section `Tutorials/Matrices/DenseMatrices` in the *Template Numerical Library Documentation* [36].

More detailed information on the structure's implementation can be found under `Namespaces/TNL/Matrices/DenseMatrix` in *Template Numerical Library Documentation* [36] and further examples of usage can be found under `Tutorials/Matrices/DenseMatrices`.

2.2 Decomposition Project

This section aims to present the project that contains the implementations of the algorithms mentioned in Sections 1.3.1 and 1.3.2. The project can be found in the *Decomposition* repository on GitLab[4]. Since the project is continually under development it is only available internally, i.e. to users belonging to the TNL group on the Mathematical Modelling Group's[5] GitLab page[6].
The Decomposition project was written in *C++* which was extended by *CUDA* (GPU programming) and *GoogleTest*[7] (unit testing - approach adopted from TNL's matrix unit tests). Additionally, *Python*[8] and *Bash*[9] were used when adopting scripts from TNL, for example, a python script that creates tables and graphs in HTML from JSON files generated by the benchmark.
First, the installation procedure and unit tests will be briefly introduced, followed by a detailed description of the implementations of the algorithms. Then, the benchmark implementation that aims to put into perspective the performance differences between the direct and iterative approaches will be presented.

Installation procedure Although development was done outside of TNL, its structuring and building processes were adopted with minor changes to allow for easier familiarization and to keep the option of incorporating the *Decomposition* project into TNL open for the future. This part will briefly describe the project's installation procedure. The prerequisites required for the installation of the project along with their recommended versions are the following:

- GCC 8.4.0 or later - for the compilation of the project.
- CUDA 11.5 or later - for computation on Nvidia GPUs.

[4]Decomposition GitLab repository URL: `https://mmg-gitlab.fjfi.cvut.cz/gitlab/tnl/lu-decomposition`
[5]MMG main page URL: `https://geraldine.fjfi.cvut.cz/mmg/index.php`
[6]TNL GitLab group URL: `https://mmg-gitlab.fjfi.cvut.cz/gitlab/tnl`
[7]GoogleTest GitHub repository URL: `https://github.com/google/googletest`
[8]Python website URL: `https://www.python.org/`
[9]Bash website URL: `https://www.gnu.org/software/bash/`

- CMake 3.20.5 or later - for building the project.
- Python 3.9 or later - optional, for using the script that transforms the benchmark results into graphs and tables.

Once all of the requirements are installed, then the installation procedure itself comprises the following steps:

1. Install the TNL library according to its documentation on the TNL website[10]. Note that for the required TNL files to be installed, it is important to use the target `all` and to include the `--install=yes` installation option.
2. Install the Decomposition project using the following command: `./install --install=yes all`. Note that the full list of compilation options can be displayed using `./install --help`.

Unit Tests Another important part of the project were the unit tests. Their structure was adopted from TNL, however, the tests themselves were slightly altered to meet the requirements of the decomposition implementations. Since the unit tests are run with every compilation of the algorithms they had to be lightweight. Specifically, the tests include the decomposition of a set of matrices with varying dimensions using the implementations. The matrix dimensions ranged from 2×2 to 38×38 and included the following: 2×2, 3×3, 4×4, 10×10, 19×19 and 38×38. The larger dimensions - from 10x10 - were chosen to test a specific part of the Iterative Crout method's implementation on the GPU: *Processing by sections* (detailed in Section 2.2.1).

2.2.1 LU Decomposition

The core of the Decomposition project is made up of the Crout method and the Iterative Crout method, detailed in Section 1.3.1 and Section 1.3.2 respectively. This part contains a description of their implementations. First, it is necessary to introduce a memory-saving concept used in this project. Then, the requirements for the implementations will be presented and, finally, the description of the implementations of the individual algorithms will follow.

Memory-saving concept Whenever LU decomposition was mentioned in the previous chapters and sections, it was presented as decomposing matrix $\mathbb{A}$ into matrices $\mathbb{L}$ and $\mathbb{U}$. However, since `DenseMatrix` stores every element of a matrix and the upper triangle of $\mathbb{L}$ and the lower triangle of $\mathbb{U}$ are zeros, then only half of the elements stored in the two matrices are used during decomposition. Therefore, to avoid using memory unnecessarily, another approach was used alongside the original: store $\mathbb{L}$ and $\mathbb{U}$ into a single matrix $\mathbb{Z}$. The lower triangle of matrix $\mathbb{Z}$ (including the main diagonal) would house the elements of $\mathbb{L}$ and the upper triangle (excluding the 1s on the main diagonal) would be made up of elements from $\mathbb{U}$.
While this approach uses less memory, it has some drawbacks:

- it is not aligned with the convention of LU decomposition to return two matrices; and

[10]TNL library installation instructions URL: `https://tnl-project.gitlab.io/tnl/#installation`

- it does not store the 1s found on the main diagonal of $\mathbb{U}$.

Thus, for the decomposed solution to be used it may be required to extract matrices $\mathbb{L}$ and $\mathbb{U}$ from matrix $\mathbb{Z}$ - this step is not implemented in the Decomposition project currently and is up to the user to handle it for now.
Nevertheless, to save memory, in this project, every decomposing method is written in two variants:

1. Decompose matrix $\mathbb{A}$ into matrices $\mathbb{L}$ and $\mathbb{U}$.
2. Decompose matrix $\mathbb{A}$ into matrix $\mathbb{Z}$ which contains elements of $\mathbb{L}$ and $\mathbb{U}$ - except for their zero-filled triangles and the main diagonal of $\mathbb{U}$.

Implementation requirements To put the algorithms mentioned earlier into the context of the project's tasks, the tasks requiring some sort of implementation were the following:

1. Implement a parallel version of LU decomposition in CUDA for GPUs.
2. Measure the acceleration of the GPU version of the LU decomposition against the CPU version.

In other words, this means implementing the following:

1. Crout method for the CPU detailed in Section 1.3.1.
2. Iterative Crout method for the GPU detailed in Section 1.3.2.

Crout method The Crout method - as described in Section 1.3.1 - was implemented only for the CPU as implementing it on the GPU would not bring any interesting results since it is a sequential algorithm. Since the implementation of the Crout method in the Decomposition project differs slightly from the pseudocode shown in Listing 1.12, the code is shown only in Attachment A.
One of the noticeable differences between the pseudocode from Listing 1.12 and the implementations is that the lines where the elements on the main diagonal of $\mathbb{U}$ are all set to 1s are not present in the latter. They were removed as it was observed that they were not needed since the algorithm would compute them regardless.

Iterative Crout method According to the requirements mentioned above, the Iterative Crout method was to be implemented only on the GPU, however, in order to verify the correctness of the results its first version was also implemented on the CPU.
The CPU version served purely as a means to verify that the GPU version did not exhibit unexpected behaviors due to parallelism, for example, race conditions. In this instance, a race condition refers to a scenario where one thread would overwrite a value before another thread would read the old value. After a brief comparison of results produced by the CPU and GPU versions on a set of matrices no differences were found. Thus, even though originally it was planned for the CPU version to mirror the GPU version and have it continually verify the results, this approach was abandoned due to the large computational time required for the CPU version to complete. In summary, the CPU version only mirrors the initial GPU version and was not developed further.

The GPU version of the Iterative Crout method went through various optimizations during development - the different concepts are detailed in Section 2.3. This section will only present the latest - and for now final - GPU version that uses matrix $\mathbb{Z}$.
The GPU implementation can be split into two parts:

1. Processing by sections.
2. Processing of a single section using the Iterative Crout method detailed in Section 1.3.2.

Processing by sections This part leverages advantages from both the sequential and parallel versions of the Crout method in order to optimize the entire procedure.
Similarly to the Iterative algorithm detailed in Section 1.3.2, the *processing by sections* concept begins by providing an initial estimate of matrix $\mathbb{Z}$, for example, $\mathbb{Z} = \mathbb{A}$. Then, $\mathbb{Z}$ is divided into submatrices of equal size (in this project, the submatrices are referred to as *sections* and are labeled with $_{sub}$, for example, $\mathbb{Z}_{sub}$). Then, beginning in the top-left corner, the sections are sequentially processed (equivalent to a submatrix of the input matrix being decomposed) using the Iterative Crout method algorithm described in Section 1.3.2. The order in which sections are processed is crucial to achieving near-optimal execution times; it is shown in Figure 2.2.

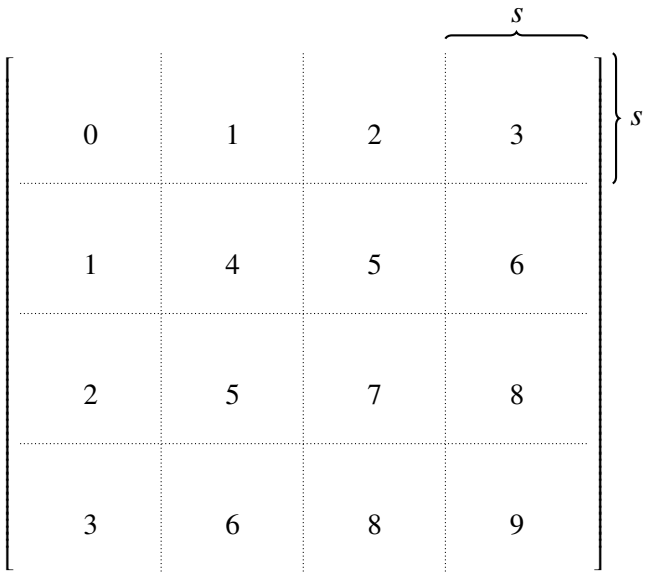


Figure 2.2: Visualization of matrix $\mathbb{Z}$ divided into sections of size $s \times s$. The order in which sections are processed is denoted by the number in each section. First, the upper-left-most section (section "0") is processed, then, the sections below and to the right of it (sections labeled with a "1") are processed in parallel. Then, certain sections to the right and below each section "1" (sections labeled with "2") are processed in parallel, etc.

The order of processing of sections is derived from the fact that a single element in $\mathbb{Z}$, for example, z_{ij} (where $i, j \in \widehat{n}$), is computed using only some elements above and to the left of it. Specifically, only $2x$ (where $x = \min(i, j)$) elements will be used, i.e. in row i only columns $[0, x]$ and in column j only rows $[0, x]$. This means that the quality of an element's value greatly depends on the quality of the elements used to compute it (quality refers to how close an element is to its final (processed) value in a

given iteration). The visualization of what elements are used to compute a single element z_{ij} is shown in Figure 2.3.

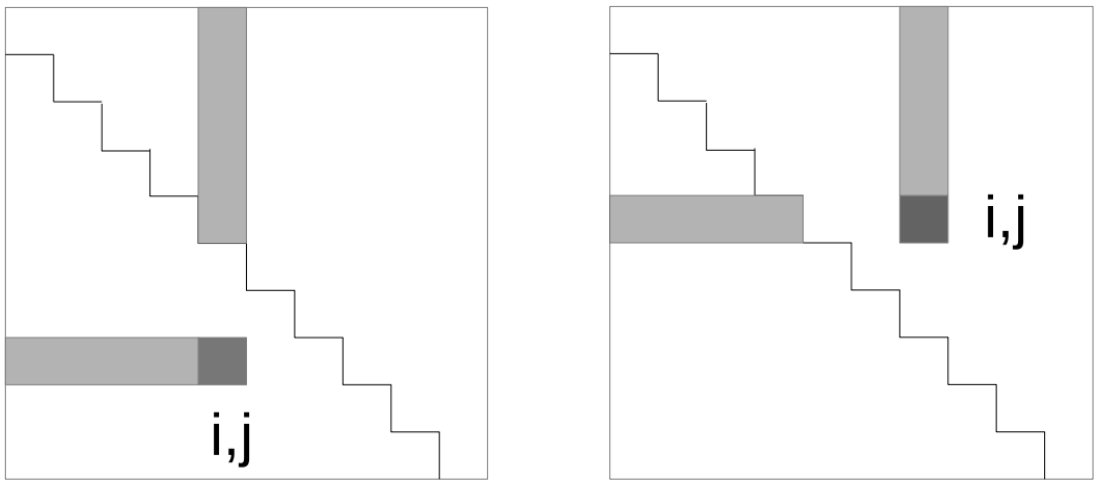


Figure 2.3: Examples showing the elements used (shaded regions) to calculate an element at row i and column j (dark square). As described earlier, only elements with indices in the interval $[0, \min(i, j)]$ are used in the element's row and column. Taken from E. Chow's and A. Patel's *Fine-Grained Parallel Incomplete LU Factorization* [38].

The implementation of the Iterative Crout method for the GPU using matrix $\mathbb{Z}$ is shown in Listing 2.2. The listing does not contain the decomposition kernel itself as it is shown later in Listing 2.3 in *Processing of a single section*.

```
1  template< typename Matrix1, typename Matrix2 >
2  void CroutMethodIterative::decompose( Matrix1& A, Matrix2& Z )
3  {
4    using Real   = typename Matrix1::RealType;
5    using Device = typename Matrix1::DeviceType;
6    using Index  = typename Matrix1::IndexType;
7
8    Index num_rows = A.getRows();
9    Index num_cols = A.getColumns();
10
11   Matrix2 Znew;
12   Znew.setLike( Z );
13   Z = A;
14   bool processed = true;
15   Real tolerance = 0;
16
17   // Compute the size of a section
18   Index sectionSize = min( max( num_cols / 10, (Index)256 ), (Index)1024 );
19   sectionSize = ( sectionSize + BLOCK_SIZE - 1 ) / BLOCK_SIZE * BLOCK_SIZE;
20
21   // Determine the number of blocks based on the section size
22   Index blocks = sectionSize / BLOCK_SIZE;
23   dim3 threadsPerBlock( BLOCK_SIZE, BLOCK_SIZE );
24   dim3 blocksPerGrid( blocks, blocks );
25
26   // Copy required data to the GPU
27   Matrix1* matrixA_kernel     = TNL::Cuda::passToDevice( A );
28   Matrix2* matrixZ_kernel     = TNL::Cuda::passToDevice( Z );
```

```
29  Matrix2* matrixZnew_kernel  = TNL::Cuda::passToDevice( Znew );
30  bool*    processed_kernel   = TNL::Cuda::passToDevice( processed );
31  Real*    tolerance_kernel   = TNL::Cuda::passToDevice( tolerance );
32
33  // Initialize the `processed` switch to page-locked host memory
34  bool* processed_host = NULL;
35  cudaMallocHost( (void **) &processed_host, sizeof(bool) );
36
37  // Allocate and initialize an array of 2 stream handles
38  int nstreams = 2;
39  cudaStream_t *streams = (cudaStream_t *)malloc( nstreams * sizeof(cudaStream_t) );
40  for( int i = 0; i < nstreams; ++i ) {
41    cudaStreamCreate( &(streams[i] ) );
42  }
43
44  // Initialize variables for keeping track of where the main diagonal section begins and ends
45  Index diag_start, diag_end, sStart, sEnd;
46
47  // Processing by sections
48  for( diag_start = 0, diag_end = min( num_cols, sectionSize ); diag_start < diag_end; diag_start += sectionSize, diag_end = min( num_cols, diag_end + sectionSize ) ) {
49    // Process the diagonal section first using the Default stream
50    do {
51      *processed_host = true;
52      cudaMemcpy( processed_kernel, processed_host, sizeof(bool), cudaMemcpyHostToDevice );
53
54      DecomposeKernel< Real, Index ><<< blocksPerGrid, threadsPerBlock >>>( matrixA_kernel, matrixZ_kernel, matrixZnew_kernel, diag_start, diag_end, diag_start, diag_end, tolerance_kernel, processed_kernel );
55
56      // Synchronize after execution of kernel before copying value of "processed"
57      cudaDeviceSynchronize();
58      cudaMemcpy( processed_host, processed_kernel, sizeof(bool), cudaMemcpyDeviceToHost );
59    } while( !*processed_host );
60
61    // Process sections below and to the right of the main diagonal section using the two streams
62    for( sStart = diag_end, sEnd = min( num_cols, diag_end + sectionSize ); sStart < sEnd;  sStart += sectionSize, sEnd = min( num_cols, sEnd + sectionSize ) ) {
63      do {
64        *processed_host = true;
65        cudaMemcpy( processed_kernel, processed_host, sizeof(bool), cudaMemcpyHostToDevice );
66
67        // Run Stream 1: iterate columns - rows should stay the same
68        DecomposeKernel< Real, Index ><<< blocksPerGrid, threadsPerBlock, 0, streams[0] >>>( matrixA_kernel, matrixZ_kernel, matrixZnew_kernel, sStart, sEnd, diag_start, diag_end, tolerance_kernel, processed_kernel );
69
70        // Run Stream 2: iterate rows - columns should stay the same
```

```
71          DecomposeKernel< Real, Index ><<< blocksPerGrid, threadsPerBlock, 0, ↩
                streams[1] >>>( matrixA_kernel, matrixZ_kernel, matrixZnew_kernel, ↩
                diag_start, diag_end, sStart, sEnd, tolerance_kernel, processed_kernel↩
                 );
72
73          // Synchronize after execution of kernels before copying value of `↩
                processed`
74          cudaDeviceSynchronize();
75          cudaMemcpy( processed_host, processed_kernel, sizeof(bool), ↩
                cudaMemcpyDeviceToHost );
76        } while( !*processed_host );
77      }
78    }
79  }
```

Listing 2.2: GPU implementation of the Iterative Crout method. Some code has been omitted as it is not important for the description, for example, checking if the input matrix is square. Overall, the code performs the following: initialization of variables (including CUDA grid specifications, page-locked host memory, streams, etc.), then the entire matrix $\mathbb{A}$ is decomposed using the *Processing by sections* concept. The constant `BLOCK_SIZE` dictates how many threads are present in a single thread block. The `processed_host` variable is purposefully initialized using page-locked host memory as its value is copied between the host and device in every iteration of the do-while loop. Thus, as described in *Page-locked host memory* in Section 1.2.3, the higher bandwidth provided can help reduce the overall time spent copying during computation. Taken from the Decomposition project repository on GitLab[4].

One of the first performance-determining aspects shown in Listing 2.2 is the size of a section (`sectionSize` or s). It affects both the speed of computation and the accuracy of results (detailed further in Chapter 3). Specifically, the phrase "accuracy of results" refers to how much the matrix resulting from $\mathbb{L} \cdot \mathbb{U}$ differs from the original input matrix $\mathbb{A}$. How the size of a section is calculated changed many times throughout development. Initially, the size of a section is set to $1/10$ of the dimensions of $\mathbb{Z}$, however, this value must be within the interval $[256, 1024]$. If it is outside of this interval, then the size is set to the interval's closest endpoint (256 or 1024). Then, its value is rounded to the nearest multiple of `BLOCK_SIZE` (8, 16, or 32) that is larger than the original `sectionSize` due to CUDA-related reasons described later in *Processing of a single section*.
In Figure 2.2 it can be seen that some sections are processed in parallel. In general, a section can be processed optimally only if specific sections above and to the left of it are already processed. The specific sections that the to-be-processed section depends on are the same as the elements required to process a specific element quickly. To imagine this better, the dark square in Figure 2.3 can be considered as the to-be-processed section and the shaded regions can be considered as the sections that must be processed before it. It is noteworthy that the general concept - how a section can be processed optimally - allows for more than two sections to be processed simultaneously. This and any other potential performance improvements are detailed in Section 2.3.
As can be seen in Listing 2.2 each section is processed using a decomposition kernel. Since each kernel is launched on a grid of blocks that are made up of threads, then each section is processed using a grid. Thus, to facilitate parallel processing of sections, CUDA streams (detailed in Section 1.2.4) were used. Specifically, sections that encompass the main diagonal (main diagonal sections) are processed using the default stream and sections processed in parallel were each given a separate stream, i.e. two streams are created and each is used to process one of the parallel sections. For example - using the numbering of sections from Figure 2.2 - first, section "0" is processed using the default stream. Then, sections labeled with a "1" are processed - each using a different stream. Following that, sections labeled with a "2" are

processed - each using a different stream - and so on until sections labeled with a "3". Then section "4" is processed using the default stream.
In summary, processing the input matrix by sections is faster than processing all of it at once since its elements - especially those in the lower-right corner - will have higher quality elements available for the computation of their value. For example, let z_{ij} (located in the lower-right corner of $\mathbb{Z}$) be the currently computed element. If the entire matrix $\mathbb{Z}$ is being processed at once and if it is a large matrix, then, since elements in the first few rows and columns (excluding the 1st row and column as their values are taken directly from $\mathbb{A}$ and not altered further) are highly inaccurate, they will cooperatively produce even less accurate elements in the later rows and columns - a snowball effect of low-quality elements. Thus, during the first few iterations, z_{ij} will be calculated using low-quality elements and it will be far from its final processed value. Slowly, through multiple iterations, the quality of elements will increase row-by-row and column-by-column which will in turn slowly process z_{ij}. Thus, when processing the entire matrix simultaneously, computing elements found further in the matrix (elements near and in the lower-right corner of the matrix) is a waste of resources and time. Ultimately, this makes processing by sections more optimal.

Processing of a single section While the previous part detailed how an entire matrix is processed by sections, this part describes how a single section is processed. A single section denotes an $s \times s$ submatrix of matrix $\mathbb{Z}$ that is labeled $\mathbb{Z}_{sub}$. Since the submatrix (section) is not being decomposed itself as a matrix, rather this part of matrix $\mathbb{Z}$ is being decomposed, then, the phrase "single section is processed" is befitting. In technical terms, this means providing a series of inputs (sub-matrices, indexes, tolerance values, etc.) to a kernel that is executed on the device. First, for the explanation of the decomposition kernel, it is necessary to slightly alter the algorithm from Section 1.3.2 - using matrix $\mathbb{Z}$ instead of $\mathbb{L}$ and $\mathbb{U}$ - to the perspective of a section:

1. Given input matrices $\mathbb{A}$ and $\mathbb{Z}^t$ calculate the next iteration of a specific submatrix: $\mathbb{Z}^{t+1}_{sub}$:

$$z^{t+1}_{ij} = a_{ij} - \sum_{k=1}^{j-1} z^t_{ik} z^t_{kj} \qquad i \geq j\,,$$

$$z^{t+1}_{ij} = \frac{1}{z^t_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} z^t_{ik} z^t_{kj} \right) \qquad i < j\,.$$

2. Check if the section has been processed according to the tolerance rule: $\left|\mathbb{Z}^{t+1}_{sub} - \mathbb{Z}^t_{sub}\right| > tol$ (where *tol* is some hard-set value, for example, 0.0001). If the tolerance rule has not passed, then go back to step 2.

3. If the tolerance rule has passed, then the algorithm has arrived at an approximate solution of a submatrix in $\mathbb{LU}$ (stored in $\mathbb{Z}$).

Note that the value of the *tol* parameter is another important factor that determines the performance (speed of execution and accuracy of results). If the value of the parameter is too high, for example, 1.5 then the algorithm will finish faster, however, the matrix resulting from $\mathbb{LU}$ will differ more significantly from $\mathbb{A}$, i.e. the decomposition will be inaccurate. On the other hand, the closer the tolerance value is to zero, the more accurate the results will be, however, it will take longer for the algorithm to finish. This parameter went through many changes, before finally settling to 0.0. The reasons behind this decision were the following:

- *Processing by sections* allows for greater granularity as the quality of each $s \times s$ section depends on the quality of specific sections that were processed before it. Thus, setting the tolerance parameter to zero means that the elements of a section are computed using elements that are as accurate as possible.
- Testing the solution on a set of matrices that consisted of 61 sparse, dense, small, and large matrices revealed that the difference in performance when setting the tolerance value near zero (for example, 0.0001) and just zero was negligible. However, the accuracy of results when $tol = 0.0$ was significantly higher - detailed further in Section 2.3 and Chapter 3.
- The testing further revealed that the accuracy of results can decrease if the dimensions of the input matrix increase. Thus, it was decided that even though the speed of execution is important, it loses meaning if the produced results are unusable.

The implementation of the decomposition kernel using matrix $\mathbb{Z}$ is shown in Listing 2.3.

```
template< typename Real, typename Index, typename Matrix1, typename Matrix2 >
__global__ void DecomposeKernel( const Matrix1* _A, Matrix2* _Z, Matrix2* _Znew, const Index col_sStart, const Index col_sEnd, const Index row_sStart, const Index row_sEnd, const Real* tolerance, bool* processed )
{
  auto& A = *_A; auto& Z = *_Z; auto& Znew = *_Znew;

  // Get a thread's submatrix ID that is offset to become a global ID of Z
  Index row = blockIdx.y * blockDim.y + threadIdx.y + row_sStart;
  Index col = blockIdx.x * blockDim.x + threadIdx.x + col_sStart;

  Index tx = threadIdx.x;
  Index ty = threadIdx.y;
  Real sum = 0;

  // Set overreaching thread ID to the closest edge of the matrix's dimensions
  Index max_col = col_sEnd - 1;
  Index max_row = row_sEnd - 1;
  Index new_row = min( row, max_row );
  Index new_col = min( col, max_col );

  // Get the smallest index of the thread's element and offset it by BLOCK_SIZE - preparation for the use of shared memory
  Index min_row_col = min( new_row, new_col ) - BLOCK_SIZE;

  // Initialize shared memory
  __shared__ Real ZsLower[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ Real ZsUpper[BLOCK_SIZE][BLOCK_SIZE];

  Index i, k;

  // Compute the sum needed for element (row, col) by loading blocks of elements from global to shared memory and multiplying them
  for( i = 0; i <= min_row_col; i += BLOCK_SIZE ) {
    ZsLower[ty][tx] = Z( new_row, i + tx );
    ZsUpper[ty][tx] = Z( i + ty, new_col );

    __syncthreads();

    #pragma unroll
```

```
37      for( k = 0; k < BLOCK_SIZE; ++k ) {
38        sum += ZsLower[ty][k] * ZsUpper[k][tx];
39      }
40      __syncthreads();
41    }
42
43    // Loop unrolling means looping by multiples of BLOCK_SIZE
44    // If the number of remaining elements is less than BLOCK_SIZE, then compute ↩
          them separately
45    ZsLower[ty][tx] = Z( new_row, min( i + tx, max_col ) );
46    ZsUpper[ty][tx] = Z( min( i + ty, max_row ), new_col );
47    __syncthreads();
48    Index t_to_use = row >= col ? tx : ty;
49    for( k = 0; k < t_to_use; ++k ) {
50      sum += ZsLower[ty][k] * ZsUpper[k][tx];
51    }
52
53    //  Terminate threads whose ID reaches out of matrix dimensions
54    if( row >= row_sEnd || col >= col_sEnd ) {
55      return;
56    }
57
58    // Compute the value of element (row, col) in the new iteration
59    if( row >= col ) {
60      Znew( row, col ) = A( row, col ) - sum;
61    } else {
62      if( Z( row, row ) == 0 ) {
63        printf( "Key element: %f\t(row:%d, col:%d)\n", Z( row, row ), (int)row, (int↩
              )col );
64        assert( Z( row, row ) != 0 );
65      }
66
67      Znew( row, col ) = ( A( row, col ) - sum ) / Z( row, row );
68    }
69
70    // Check if the element (row, col) has been processed
71    if( abs( Znew( row, col ) - Z( row, col ) ) > *tolerance ) {
72      *processed = false;
73    }
74
75    // Assign element for this iteration
76    Z( row, col ) = Znew( row, col );
77  }
```

Listing 2.3: The constant `BLOCK_SIZE` is by default hard-coded to be equal to 8 in the `CroutMethod Iterative` header file using the `#define` macro. The kernel takes matrices $\mathbb{A}$, $\mathbb{Z}$ (iteration t of $\mathbb{Z}$), and $\mathbb{Z}_{new}$ (iteration $t+1$ of $\mathbb{Z}$) as input parameters, and uses other input variables to offset the threads in the grid so that only a specific section of $\mathbb{Z}$ is computed. Note that the kernel is not part of the `CroutMethod Iterative` class since kernels cannot be members of a class - as mentioned in *Function extensions* in Section 1.2.5 - however, it is located in the same file that it is called from. Taken from the Decomposition project repository on GitLab[4].

The kernel presented in Listing 2.3 has the following input parameters:

- `col_sStart` - Starting column index of the section in matrix $\mathbb{Z}$.

- `col_sEnd` - Ending column index of the section in matrix $\mathbb{Z}$.
- `row_sStart` - Starting row index of the section in matrix $\mathbb{Z}$.
- `row_sEnd` - Ending row index of the section in matrix $\mathbb{Z}$.
- `tolerance` - Tolerance parameter stored in the global memory of the device. All threads only need to read it to check the tolerance rule for their element.
- `processed` - Boolean switch stored in the global memory of the device that determines whether the element `(row, col)` has been processed. At the end of the kernel, each thread evaluates whether its element has been processed according to the tolerance rule described above. While it is generally considered bad practice for multiple threads to write to a single address in memory, in this instance, is it befitting. The reasoning is that for the algorithm to perform another iteration, only one thread needs to set `processed = false`. As long the value of `processed` is copied from the device to the host after all threads have finished executing the kernel, then the intended behavior occurs. This is assured using the `cudaDeviceSynchronize()` function used in lines 57 and 74 in Listing 2.2 as explained in *Explicit synchronization* in Section 1.2.4.

From the parameters listed above, the first four are used to determine where in matrix $\mathbb{Z}$ the thread's element is located. Together, the threads then process the elements of the intended section. The remaining input parameters are specifically prefixed with a `_` as they are not used, instead, pointers to their memory address are initialized in such a way that allows for clean use of the overloaded `operator()` which is detailed *Dense matrix* in Section 2.1.
Furthermore, other than the input parameters and CUDA-related extensions detailed in *Inter-kernel Extensions* in Section 1.2.5 the kernel also has access to the macro-defined `BLOCK_SIZE` variable. The `BLOCK_SIZE` constant is set using the `#define` macro to make the compiler unroll the `for` loop during compile time using `#pragma unroll`. Loop unrolling is a compiler optimization that reduces the work done by the processing thread. Specifically, it removes the need for the processing thread to initialize the loop variable, evaluate the condition and increment the variable [39]. However, if the number of loops in a `for` loop is now known during compile time, then the loop cannot be unrolled. An example showing loop unrolling is shown in Listing 2.4.

```
1  // Hard-set constant - value is known by the compiler at compile time
2  #define BLOCK_SIZE 8
3
4  // Original for loop
5  for( k = 0; k < BLOCK_SIZE; ++k ) {
6    sum += ZsLower[ty][k] * ZsUpper[k][tx];
7  }
8
9  // For loop transformed using loop unrolling during compilation
10 sum += ZsLower[ty][0] * ZsUpper[0][tx];
11 sum += ZsLower[ty][1] * ZsUpper[1][tx];
12 sum += ZsLower[ty][2] * ZsUpper[2][tx];
13 sum += ZsLower[ty][3] * ZsUpper[3][tx];
14 sum += ZsLower[ty][4] * ZsUpper[4][tx];
15 sum += ZsLower[ty][5] * ZsUpper[5][tx];
16 sum += ZsLower[ty][6] * ZsUpper[6][tx];
17 sum += ZsLower[ty][7] * ZsUpper[7][tx];
```

Listing 2.4: Example demonstrating loop unrolling. In this specific case, if loop unrolling is used, then the processor will not have to: initialize `k`, evaluate `k < BLOCK_SIZE` nine times, and increment `k` eight times. If `BLOCK_SIZE` would be obtained as an input parameter, then the loop would not be unrolled as the input parameter is not known during compile time. Taken from *Embedded Computing for High Performance* [39].

Another important aspect of the implementation lies at the core of the kernel: the use of shared memory. Since each section is assigned a grid made up of blocks, then each block has limited shared memory available to it as described in *Shared memory* in Section 1.2.3. Overall, shared memory is used exactly as described in the matrix multiplication *Example with Shared Memory* in Section 1.2.6 with an exception. In the example - depicted in Figure 1.16 - the blocks loop through the entire matrix. As explained earlier in *Processing by sections* in Section 2.2.1 and shown in Figure 2.3 using all elements in the row and column of an element is not required. Thus, in this implementation, the thread block finishes looping once the last required block of elements is used for the computation. However, this presented an obstacle: once the thread block arrives at the last block of elements required for the computation, each of its threads must stop at different elements. In other words, the number of loops required for each thread in the last block of elements is not known at compile time. This means that loop unrolling is not possible for the last block of required elements and it can only be performed for the blocks before. For this purpose, the variable `min_row_col` is offset by `BLOCK_SIZE` to make the loop using loop unrolling end sooner. The remaining elements are then computed using a regular `for` loop (lines 43 - 51 in Listing 2.3).
As detailed in *Shared memory* in Section 1.2.3 when using shared memory, it is important to avoid a bank conflict. In the kernel shown in Listing 2.3 shared memory is only read from on lines 38 and 50. Specifically, in the multiplication operation `ZsLower[ty][k] * ZsUpper[k][tx]` shared memory is being read from in two different ways (also detailed in *Shared memory* in Section 1.2.3):

- `ZsLower[ty][k]` - All threads in a warp have the same `ty` id. This means that for a given `k` all threads of a warp access the same location in shared memory - known as a *Broadcast* - which is a fast data-serving strategy.
- `ZsUpper[k][tx]` - Each thread in a warp has a unique and linearly increasing `tx`. This means that for a given `k` each thread of a warp accesses one word in a different bank. In other words, a single word is requested from each bank by a unique thread, thus, all 32 threads of a warp are served by all 32 banks simultaneously and expeditiously.

In *Processing by sections*, it was mentioned that the size of a section must be a multiple of `BLOCK_SIZE`. Although `sectionSize` is not used directly in the kernel, the use of shared memory needed to be explained in order to present the reasoning for this limitation. Lines 53 - 56 in Listing 2.3 show that any threads whose shifted global ID overreaches the matrix dimensions are terminated. In a typical CUDA kernel, this step would be performed near the beginning of the kernel. However, in this case, the threads that overreach are needed for loading data from global to shared memory. For example, looking at Figure 1.16, if the matrix dimensions were not a multiple of `BLOCK_SIZE` and the blocks computing the last columns were partly outside of matrix dimensions, then the right-most threads of the upper block (matrix $\mathbb{B}$) would reach out of matrix dimensions and they should - under normal circumstances - be terminated to avoid erroneous behavior. However, the same threads are concurrently being used to load data from global to shared memory in the left block (matrix $\mathbb{A}$) - there the threads would not reach out of bounds. For this reason, the threads cannot be terminated when computing the sum and that is also the reason

why their cut global ID is used during calculation. The threads mentioned compute sums identical to the threads whose global ID is on the edge of the matrix's dimensions. Then, the values computed by the cut threads are discarded once they are terminated. Thus, if `sectionSize` is not a multiple of `BLOCK_SIZE`, then loop unrolling would not be possible as the block of elements loaded into shared memory would not cover the entire section.

2.2.2 Benchmark

As mentioned in *Implementation requirements* in Section 2.2.1 one of the goals of this project was to measure the acceleration of the GPU version of the LU decomposition against the CPU version. For this purpose, a benchmark system was implemented. In its majority, the structuring and code were adopted from the SpMV TNL Benchmark structure[11] which was implemented by one of the main developers for TNL, Ph.D. student Jakub Klinkovský. However, some logic was reworked specifically for this project. The benchmarking procedure is initiated from a Bash script[12] and it involves running the entire benchmark six times. Specifically, for each precision (single or double) it is run three times - each with a different number of threads per block: 8, 16, or 32; i.e. blocks of 8×8, 16×16, or 32×32 threads. More threads per block are not possible since the maximum number of threads per block is 1024 - detailed in *Block* in Section 1.2.2. While the "threads-per-block" part of the benchmark was initially added to determine the optimal number of threads per block, it was never apparent from any benchmark run which value is optimal, therefore, it was kept for future development. Each run of the benchmark involves decomposing a set of matrices; each matrix is stored in a `.mtx` file. Specifically, the subroutine for benchmarking a matrix is as follows (assuming the `precision` and `threads-squared-per-block` arguments are set):

1. Load the matrix from its `.mtx` file using the `MatrixReader` class implemented in TNL[13] to an instance of `DenseMatrix`.
2. Compute the decomposition using the CPU implementation of the Crout method described in *Crout method* in Section 2.2.1 and, if specified, compute the maximum difference of $|\mathbb{A} - \mathbb{LU}|$. Then, log statistics such as execution time, bandwidth achieved, maximum difference, etc. into the log file.
3. Compute the decomposition using the GPU implementation of the Iterative Crout method described in *Iterative Crout method* in Section 2.2.1 and, if specified, compute the maximum difference of $|\mathbb{A} - \mathbb{LU}|$. Then, log statistics such as execution time, bandwidth achieved, maximum difference, etc. into the log file.

Additionally, the log file can be transformed into an HTML table and data such as bandwidth and relative speedup can be plotted into graphs using a convenient Python script[14] - originally adopted from TNL and later modified.

[11] TNL SpMV Benchmark GitLab repository URL: `https://gitlab.com/tnl-project/tnl/-/tree/main/src/Benchmarks/SpMV`

[12] Benchmark script URL in the Decomposition GitLab repository: `https://mmg-gitlab.fjfi.cvut.cz/gitlab/tnl/lu-decomposition/-/blob/master/src/Benchmark/scripts/run-decomposition-benchmark`

[13] `MatrixReader` class header file in the TNL GitLab repository URL: `https://gitlab.com/tnl-project/tnl/-/blob/main/src/TNL/Matrices/MatrixReader.h`

[14] Transforming Python script GitLab repository URL: `https://mmg-gitlab.fjfi.cvut.cz/gitlab/tnl/lu-decomposition/-/blob/master/src/Benchmark/scripts/decomposition-benchmark-make-tables-json.py`

As mentioned in Section 3.2 of the author's bachelor thesis *Formats for storage of sparse matrices on GPU* [1], benchmarks can be used to identify issues missed by unit tests. This claim persisted during the development of this project as some matrices were too large to be stored using `DenseMatrix` on the GPU. Furthermore, the benchmark implementation played a crucial role when optimizing the solution.

2.3 Optimization

In the preceding sections, it was mentioned that the *Decomposition* project went through various optimizations during development. This section aims to summarize the evolution of the GPU LU decomposition implementation from its initial naive version to its present form. Additionally, a selection of concepts that were attempted, but ultimately ended up being suboptimal will also be described. The performance of each change was measured by running the benchmark on a set of 50 sparse and dense matrices with varying dimensions between 27×27 and $10,581 \times 10,581$. This section will not present specific results (execution time, bandwidth, etc.), nor will some of the matrices will be presented or described in greater detail; that will be shown in Chapter 3.

Initial naive implementation The initial naive implementation was designed in such a manner that the entire matrix was decomposed by a single CUDA grid. In other words, each thread would compute one element in each iteration, then, the tolerance rule would be imposed (with $tol = 0.001$). The kernel and how the processing was handled within `CroutMethodIterative::decompose()` are shown in Listing 2.5.

```
1  template< typename Real, typename Index, typename Matrix1, typename Matrix2 >
2  __global__ void DecomposeKernel( const Matrix1* A, Matrix2* Z, Matrix2* Znew,
       Index row_size )
3  {
4    int row = blockIdx.y * blockDim.y + threadIdx.y;
5    int col = blockIdx.x * blockDim.x + threadIdx.x;
6
7    if( row < row_size && col < row_size ) {
8      Index k;
9      Real sum = 0;
10
11     if( row >= col ) {
12       sum = 0;
13       for( k = 0; k < col; k++ ) {
14         sum = sum + Z->getElement( row, k ) * Z->getElement( k, col );
15       }
16
17       Znew->setElement( row, col, A->getElement( row, col ) - sum );
18     }
19     else {
20       TNL_ASSERT( Z->getElement( row, row ) != 0, std::cerr << "Z( " << row << ",
           " << row << " ) = 0. Cannot divide by 0!" << std::endl );
21       sum = 0;
22       for( k = 0; k < row; k++ ) {
23         sum = sum + Z->getElement( row, k ) * Z->getElement( k, col );
24       }
25
26       Znew->setElement( row, col, ( A->getElement( row, col ) - sum ) / Z->
             getElement( row, row ) );
```

```
27      }
28    }
29  }
30
31  template< typename Matrix1, typename Matrix2 >
32  void CroutMethodIterative::decompose( Matrix1& A, Matrix2& Z )
33  {
34    // ...
35    do {
36      DecomposeKernel< RealType, IndexType, Matrix1, Matrix2 ><<< blocksPerGrid, ↩
          threadsPerBlock >>>( matrix1_kernel, matrix2_kernel, matrix2new_kernel, ↩
          num_rows );
37
38      // Determine if Z has been processed by computing the maximum absolute ↩
          difference between Z and Znew, and comparing it with a hard-set tolerance
39      processed = matrixDifferenceTolerable( Znew, Z );
40
41      Z = Znew;
42    } while( !processed );
43    // ...
44  }
```

Listing 2.5: Initial naive implementation of the GPU version of the Iterative Crout method. Taken from the Decomposition project repository on GitLab[4].

As can be seen in Listing 2.5, the implementation had several drawbacks:

- The condition on line 7 caused a divergence of threads (described in *Warp* in Section 1.2.2). However, this affected only the warps whose threads would overreach the matrix dimensions.
- The condition on line 11 also caused a divergence of threads. However, in this instance, the majority of warps were affected.
- All operations with elements of the input matrices are done in the device's global memory. Reading from global memory is suboptimal compared to reading from shared memory, however, global memory bandwidth is even lower if the memory access does not coalesce. Since the number of loops each thread performed in its `for` differed, then further thread divergence and non-coalesced memory access occurred as a consequence.
- The inefficient processing described at the end of *Processing by sections* in Section 2.2.1 is precisely the means of processing used in this naive implementation.

In summary, this implementation was suboptimal, however, it served as a foundation for future development.

Dynamic parallelism Following the initial naive implementation, it was theorized that the suboptimal performance stemmed from repeatedly copying between device and host. The reasoning behind this thought came from the fact that the tolerance rule was checked on the host (line 39 in Listing 2.5). Practically, this meant that matrices `Z` and `Znew` were copied between the host and device in every iteration. Dynamic parallelism[15] was found as a potentially suitable solution. The solution involved having

[15]CUDA Dynamic Parallelism API and Principles URL: `https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles/`

one main kernel, that would then house the tolerance-checking `do-while` cycle which would launch another kernel from within itself. However, this solution ultimately proved suboptimal and as such, it was abandoned in favor of a multiple kernel setup.

Multiple kernels The multiple kernel setup involved splitting the logic done on the host (checking if the matrix is processed and assigning the newly-computed value of `Z` to `Znew`) into two kernels:

- `ProcessedKernel` - Kernel that would check whether `Z` has been processed according to the tolerance rule.
- `MatrixAssignKernel` - Kernel that would assign `Z` to `Znew`.

The three kernels would then be called in every iteration of the `do-while` cycle - shown in Listing 2.6.

```
1  do
2  {
3    *processed_host = true;
4    cudaMemcpy( processed_kernel, processed_host, sizeof(bool), ↩
         cudaMemcpyHostToDevice );
5
6    DecomposeKernel< RealType ><<< blocksPerGrid, threadsPerBlock >>>( ↩
         matrixA_kernel, matrixZ_kernel, matrixZnew_kernel, num_rows );
7
8    ProcessedKernel< RealType ><<< blocksPerGrid, threadsPerBlock >>>( ↩
         matrixZ_kernel, matrixZnew_kernel, num_rows, processed_kernel );
9
10   MatrixAssignKernel<<< blocksPerGrid, threadsPerBlock >>>( matrixZ_kernel, ↩
         matrixZnew_kernel, num_rows );
11
12   cudaMemcpy( processed_host, processed_kernel, sizeof(bool), ↩
         cudaMemcpyDeviceToHost );
13 } while( !*processed_host );
```

Listing 2.6: Solution that replaced the procedure presented on lines 31 - 44 in Listing 2.5. Taken from the Decomposition project repository on GitLab[4].

Ultimately this solution improved the performance over both the *Initial naive implementation* and the failed solution involving *Dynamic parallelism*.

Calculation by tiles using shared memory The core of the decomposition kernel from the multiple kernel setup was still the same as in the *Initial naive implementation* (lines 7 - 28 in Listing 2.5). Thus, to improve performance, the core logic was reworked. Since the algorithm for computing LU decomposition is similar to matrix multiplication - with minor differences - the logic from CUDA's *Example with Shared Memory* shown in Listing 1.11 was adopted. Listing 2.7 shows the replacement for the lines mentioned earlier.

```
1  int tx = threadIdx.x;
2  int ty = threadIdx.y;
3
4  RealType sum = 0;
5
6  for( int i = 0; i <= num_rows; i += BLOCK_SIZE ) {
```

```
7     __shared__ RealType ZsLower[BLOCK_SIZE][BLOCK_SIZE];
8     __shared__ RealType ZsUpper[BLOCK_SIZE][BLOCK_SIZE];
9
10    // Possible non-coalesced global memory access and thread divergence
11    ZsLower[ty][tx] = ( row >= num_rows || i + tx >= num_rows ) ? 0 : Z->↩
          getElement( row, i + tx );
12    ZsUpper[ty][tx] = ( col >= num_rows || i + ty >= num_rows ) ? 0 : Z->↩
          getElement( i + ty, col );
13
14    // Synchronize to make sure the matrices are loaded
15    __syncthreads();
16
17    // Get global ID of max current position
18    int glob_id = i + BLOCK_SIZE;
19
20    if( row >= col ) {
21      // If row < BLOCK_SIZE or col < BLOCK_SIZE, then use the for loop without ↩
            pragma unroll
22      if( col >= glob_id ) {
23        #pragma unroll
24        for( int k = 0; k < BLOCK_SIZE; k++ ) {
25          sum += ZsLower[ty][k] * ZsUpper[k][tx];
26        }
27      } else {
28        for( int k = 0; k < tx; k++ ) {
29          sum += ZsLower[ty][k] * ZsUpper[k][tx];
30        }
31        break;
32      }
33    } else {
34      if( row >= glob_id ) {
35        #pragma unroll
36        for( int k = 0; k < BLOCK_SIZE; k++ ) {
37          sum += ZsLower[ty][k] * ZsUpper[k][tx];
38        }
39      } else {
40        for( int k = 0; k < ty; k++ ) {
41          sum += ZsLower[ty][k] * ZsUpper[k][tx];
42        }
43        break;
44      }
45    }
46
47    // Synchronize to make sure that the preceding computation is done before ↩
          loading two new submatrices in the next iteration
48    __syncthreads();
49  }
50
51  // Terminate overreaching threads
52  if( row >= num_rows || col >= num_rows ) {
53    return;
54  }
55
56  // Compute the element (row, col)
57  if( row >= col ) {
58    Znew->setElement( row, col, A->getElement( row, col ) - sum );
59  } else {
60    if( Z->getElement( row, row ) == 0 ) {
```

```
61      printf( "Key element: %f\n", Z->getElement( row, row ) );
62      printf( "row:%d\tcol:%d\n", row, col );
63      assert( Z->getElement( row, row ) != 0 );
64    }
65    Znew->setElement( row, col, ( A->getElement( row, col ) - sum ) / Z->↩
          getElement( row, row ) );
66  }
```

Listing 2.7: Calculation of a single iteration of the GPU Iterative Crout method using logic from Listing 1.11. Taken from the Decomposition project repository on GitLab[4].

While the procedure shown in Listing 2.7 removed some of the inefficiencies that the previous solution posed, it still contained some undesired behaviors. For example, the nested conditions that made sure each thread stopped its computation at the desired element. The conditions could cause thread divergence in almost any warp. Nevertheless, the reworked part of the implementation performed better than its predecessor and it was accepted as an optimization.

Elimination of conditions As mentioned in *Calculation by tiles using shared memory*, one of the suboptimal aspects of the decomposition kernel's core was the handling of conditions. To solve this issue, another approach was devised that would almost entirely eliminate the need for conditions. Specifically, lines 6 - 49 in Listing 2.7 were replaced with the code shown in Listing 2.8.

```
1   int max_id = num_rows - 1;
2   int new_row = min( row, max_id );
3   int new_col = min( col, max_id );
4
5   // Each thread needs to end its calculation by its smallest dimension
6   // Offest by BLOCK_SIZE, so in the for loop it is not needed to check `i + ↩
        BLOCK_SIZE <= min_row_col` in every iteration
7   int min_row_col = min( new_row, new_col ) - BLOCK_SIZE;
8
9   // Allocate Lower and Upper submatrices from matrix Z into which shared memory ↩
        will be loaded.
10  __shared__ RealType ZsLower[BLOCK_SIZE][BLOCK_SIZE];
11  __shared__ RealType ZsUpper[BLOCK_SIZE][BLOCK_SIZE];
12
13  // Initialize loop variable outisde of loop
14  int i;
15
16  // End the for loop early if this thread needs to end its calculation in the ↩
        current loop's submatrix
17  // The original condition in the for loop was: `i + BLOCK_SIZE <= min_row_col`
18  // Since all threads in a block either need to end their calculations in a ↩
        submatrix, or they all do not
19  // Submatrices are iterated until the loops gets to a submatrix where the ↩
        threads would need to end their calculations
20  // This means that the threads would need to end their calculation before ↩
        BLOCK_SIZE and the pragma loop could not be used In that case, a loop ↩
        specific is needed for that thread
21  for( i = 0; i <= min_row_col; i += BLOCK_SIZE ) {
22    ZsLower[ty][tx] = Z( new_row, i + tx );
23    ZsUpper[ty][tx] = Z( i + ty, new_col );
24
25    // Synchronize to make sure the matrices are loaded
26    __syncthreads();
```

```
27
28      #pragma unroll
29      for( int k = 0; k < BLOCK_SIZE; ++k ) {
30        sum += ZsLower[ty][k] * ZsUpper[k][tx];
31      }
32
33      // Synchronize after computation before loading new submatrices
34      __syncthreads();
35    }
36
37    // Perform calculations remaining for each thread
38    // Check that threads would not reach out of range when loading data using min
39    ZsLower[ty][tx] = Z( new_row, min( i + tx, max_id ) );
40    ZsUpper[ty][tx] = Z( min( i + ty, max_id ), new_col );
41
42    // Synchronize the threads before working with data loaded from global memory
43    __syncthreads();
44
45    int t_to_use = row >= col ? tx : ty;
46    for( int k = 0; k < t_to_use; ++k ) {
47      sum += ZsLower[ty][k] * ZsUpper[k][tx];
48    }
```

Listing 2.8: Core of the decomposition kernel with the inefficient conditions replaced by a combination of the `min()` function and helpful variables. Taken from the Decomposition project repository on GitLab[4].

This approach further improved the performance and thus it was accepted as an optimization. Ultimately, except for combining all three kernels into one, this was the final change before the final implementation - *Processing by sections* - was introduced.

Note that up until this point the tolerance value was set to 0.001, however, with the introduction of *Processing by sections* the accuracy of results was inadequate. Thus, through testing, it was observed that 0.0 was the optimal value as it greatly improved the accuracy of results while only slightly reducing the speed of execution.
Additionally, the interval for the size of the section in the first version of *Processing by sections* was $[512, 1024]$. While discovering the optimal interval range, it was observed that even with the tolerance value set to zero, if the section size was decreased to less than 256×256 then the results produced were less accurate; this behavior remains unexplained. Theoretically, since the zero-tolerance rule leaves no room for errors, it should not matter what the section size is. On the other hand, it is possible that there was an error somewhere in the implementation, however, it has not been found yet.

Other explored improvements After arriving at the final implementation, two potential improvements were explored:

1. Setting the size of a section to $1/2$ of the maximum number of active threads permitted by the GPU.
2. Processing by anti-diagonal sections.

Section size dependent on maximum active threads The first explored improvement aimed to tailor the solution to the capabilities of the GPU used for the computation. The concept of the maximum

number of active threads was explained in *Grid* in Section 1.2.2 and in *Concurrent kernel execution* in Section 1.2.4. The maximum number of active threads for a specific GPU can be obtained using the following formula:

$$max\_active\_threads = sm\_count \cdot max\_threads\_per\_sm\,,$$

which can be implemented using the CUDA `cudaDeviceProp`[16] structure as shown in Listing 2.9.

```
int cuda_device = 0;
cudaDeviceProp deviceProp;
cudaGetDevice( &cuda_device );
cudaGetDeviceProperties( &deviceProp, cuda_device );

int maxActiveThreads = deviceProp.multiProcessorCount * ↩
    deviceProp.maxThreadsPerMultiProcessor;

// Two (sectionSize x sectionSize) sections will be running in parallel at most
int sectionSize = sqrt( maxActiveThreads / 2 );
// Round sectionSize to a multiple of BLOCK_SIZE
sectionSize = ( sectionSize / BLOCK_SIZE ) * BLOCK_SIZE;
```

Listing 2.9: Code detailing the calculation of the size of a section based on the maximum number of active threads for a given GPU. The structure `cudaDeviceProp` contains properties and information about the given CUDA device. Taken from the Decomposition project repository on GitLab[4].

While this change significantly improved the speed of execution, the results it produced were less accurate than that of the final implementation. It seems that the same behavior as when `sectionSize` was set below 256 - described at the end of *Elimination of conditions* - occurred. Since a solution to the problem of inaccuracy was not found, this change was not accepted.

Processing by anti-diagonal sections At the end of *Processing by sections* in Section 2.2.1, it was mentioned that more than two sections can be processed simultaneously. Furthermore, this can be done while still adhering to the general rule of optimal processing that states: "*certain sections above and to the left of a section must be processed for the section to be processed optimally*". In this instance "optimally" means quickly and accurately. Following the rule of optimal processing, Figure 2.4 shows an alternative order of processing: *Processing by anti-diagonal sections*.

Theoretically, this order of processing allows for more parallelization than the final implementation as more independent elements are processed at once. However, practically, the implementation was not as performant as the final implementation (detailed in *Iterative Crout method* in Section 2.2.1). The reason for this was that the indexing system was too complex and required more overhead computation. Further attempts at lowering the overhead computations were not successful and as such, the solution was put aside. Note that in the attempted implementation all sections in an anti-diagonal were iterated until all of them have been processed. This means that even if one section has been processed, it was still being computed over and over if another section in the same anti-diagonal was not yet processed. Furthermore, the order of execution pictured in Figure 2.4 does not take into account all possible orders of execution. For example, the left-most section labeled with a "4" can be processed before the section above it, since

[16] `cudaDeviceProp` Struct Reference URL: `https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html`

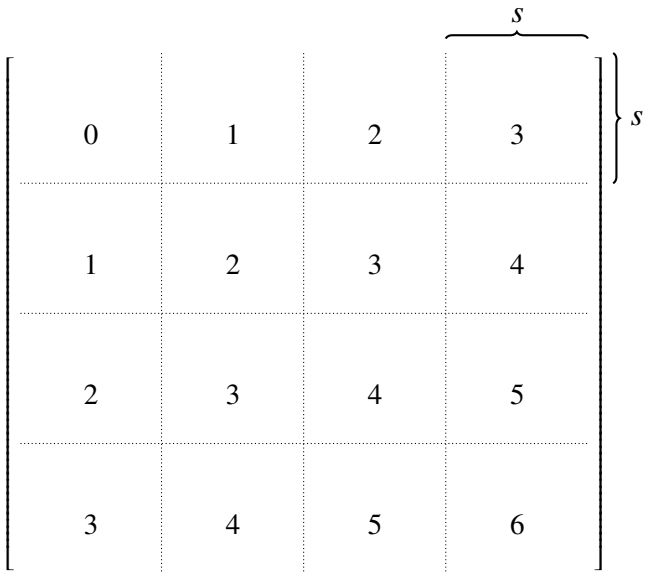


Figure 2.4: Visualization of matrix $\mathbb{Z}$ divided into sections of size $s \times s$. The order in which sections are processed is denoted by the number in each section. First, the upper-left-most section (section "0") is processed, then, the sections below and to the right of it (sections labeled with a "1") are processed in parallel. Then, all sections below and to the right of each section labeled with a "1" are processed (sections labeled with a "2") in parallel, and so on.

the elements in the former only use the elements from sections "1" and "2" in the same column and section "3" in the same row.

The solution to this problem would be to implement a more complex order execution system, which would keep track of which sections can be processed based on whether the sections it depends on have already been processed. In other words, as soon as all sections a section depends on would be processed, then the processing of this section would be immediately scheduled on the GPU.
Another advantage of such a system would be its future-proof nature. In the final implementation, `sectionSize` is restricted to the interval $[256, 1024]$. The number of threads allocated by the final implementation is more than most modern GPUs are capable of actively running. However, if future GPUs are capable of running more threads at once than required by the final implementation, then the GPU's resources would be under-utilized since sections that can be processed would be waiting until they are scheduled for execution. On the other hand, a solution involving the order execution system would allocate as many threads as possible without compromising the quality of processed elements. While such a system has neither been designed nor implemented, it is a possible feature to consider for future development.

Chapter 3

Comparing Decomposition Implementations

In Section 2.2.2 the benchmark structure implemented in order to compare the performance of the GPU version of the LU decomposition and the CPU version was presented. This chapter will present and analyze the results of the benchmark that was run in order to compare the aforementioned implementations. To begin with, the benchmarked implementations will be briefly summarized along with the specifications of the platform that the benchmarks were run on. Then, the set of matrices used in the benchmarks will be listed. Finally, the results of the benchmark will be presented and analyzed.

3.1 Benchmarked Implementations

The implementations in the *Decomposition* project that were benchmarked are the following:

- *Crout method* (CM) - This implementation is only available for the CPU. At its core, it is a sequential algorithm that delivers a decomposition of the input matrix in one pass.
- *Iterative Crout method* (ICMx) - This implementation is available only for the GPU. At its core, it is an iterative and parallelizable algorithm that converges to an approximate decomposition of the input matrix. This implementation was further divided into three different configurations of threads per block (denoted by x): 8×8 (ICM8), 16×16 (ICM16), and 32×32 (ICM32); ICMx refers to all three configurations.

3.2 Benchmark Platform Specifications

The benchmark was run on the RCI (Research Center for Informatics in CTU Prague) Cluster[1] which is an HPC (High-Performance Computing) infrastructure made up of CPU, GPU, and SMP nodes [40]. The hardware and software specifications of the cluster node that the benchmarks were run on can be found in Table 3.1.

[1] RCI homepage URL: `http://rci.cvut.cz/`

CPU	AMD EPYC 7543 @ 3.1GHz (32 cores, 64 threads)
RAM	32GB RAM
GPU	Nvidia Tesla A100 40GB HBM2 (1.6 TByte/s memory bandwidth)
Operating System	CentOS 8 Linux
Compiler	GCC 10.3
CUDA	CUDA 11.4.1

Table 3.1: Specifications of the `gpu` RCI Cluster node that the benchmarks were run on. Taken from *RCI Cluster Hardware* [40] and the configuration of the computation job.

3.3 Matrices Used for Benchmarks

The benchmarks for the implementations were run on a set of 63 matrices. The smaller number of matrices used for this benchmark is due to the fact that the current version of the *Decomposition* project does not support the decomposition of matrices that are not strongly regular, i.e. matrices that require a permutation matrix for a successful decomposition - described in Section 1.3. This means that any matrices used in the benchmark had to be checked for the aforementioned requirement which made the selection process lengthy.

The majority of matrices from the set used are sparse (obtained from the *SuiteSparse Matrix Collection* [41]). The remaining matrices are dense (randomly generated using the Python script shown in Listing B.1 in Attachment B). The dimensions of the matrices in the set ranged from 27×27 to $10,793 \times 10,793$ and, in general, their nonzero elements were mostly located along the main diagonal with some exceptions. For the direct comparison of results, 14 matrices were selected from the set of 63. The chosen matrices were selected to represent a wide variety of characteristics: density/sparsity, nonzero element structure, and dimensions. The 14 selected matrices can be found in Table 3.2.

Matrix	Rows	Columns	Nonzeros	Avg. nonzeros per row
bcsstk03	112	112	640	5.7
494_bus	494	494	1,666	3.4
LeGresley_2508	2,508	2,508	16,727	6.7
Cejka2842	2,842	2,842	8,076,964	2,842.0
rail_5177	5,177	5,177	35,185	6.8
c-31	5,339	5,339	78,571	14.7
s3rmt3m3	5,357	5,357	207,123	38.7
s1rmq4m1	5,489	5,489	262,411	47.8
Na5	5,832	5,832	305,630	52.4
Cejka5943	5,943	5,943	35,319,249	5,943.0
fp	7,548	7,548	834,222	110.5
Cejka7580	7,580	7,580	57,456,400	7,580.0
bundle1	10,581	10,581	770,811	72.8
Cejka10793	10,793	10,793	116,488,849	10,793.0

Table 3.2: Set of 14 selected matrices from the *The university of Florida sparse matrix collection* [41] and randomly generated dense matrices (labeled *Cejka<num_rows>*) that was used for the direct comparison of implementations.

3.4 Benchmark Results

This section will present and analyze the benchmark results obtained from decomposing the matrices shown in Table 3.2 using the implementations mentioned in Section 3.1 on the platform described in Section 3.2. The benchmark subroutine described in Section 2.2.2 assumed that each matrix is decomposed once, however, in this benchmark run, in order to minimize anomalous behaviors and assure the quality of presented results, each matrix was decomposed 100 times. Then, the data collected across all 100 runs (execution time, speedup, etc.) was averaged and subsequently logged. First, the comparison of speedup between CM and ICM*x* will be detailed for both single and double precision. Then, the speedup comparison across the entire set of 63 matrices will be briefly shown, along with the accuracy of results for all matrices. Finally, the benchmark results measured during development for the optimizations described in Section 2.3 will be shown. Full benchmark results - including logs - are available upon request.

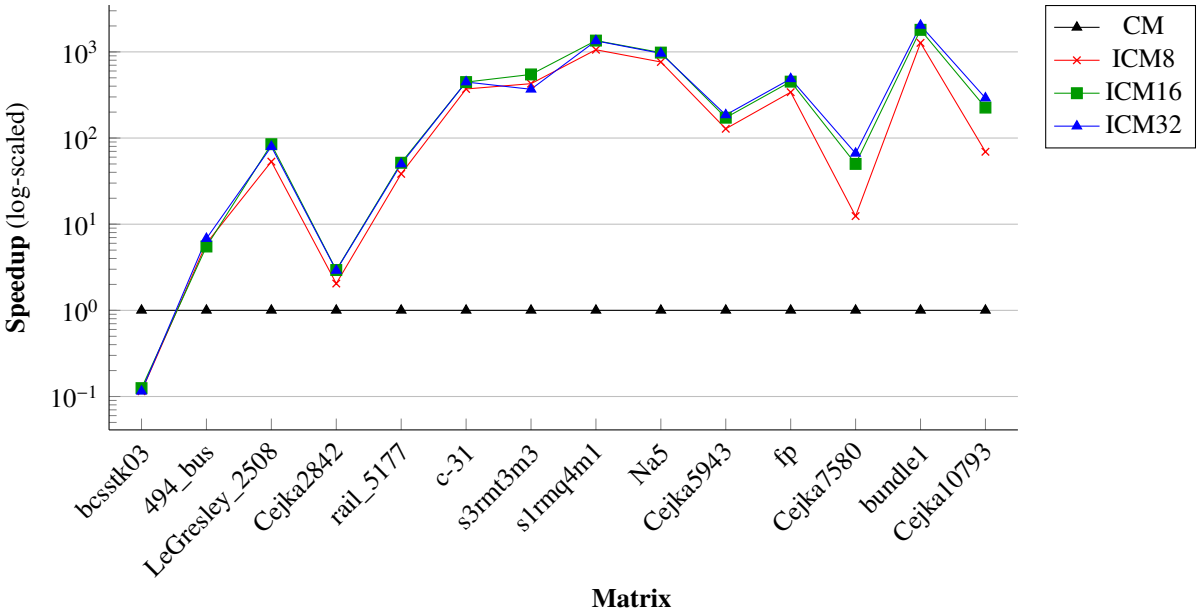
Note that the term *time*, or *execution time*, represents the execution time of the entire `CroutMethodIterative::decompose()` method. In other words, not only the execution time of the kernel but also the set up of the computation and copying of the required data from the host to the device.

3.4.1 Speedup Comparison Between CM and different ICMs

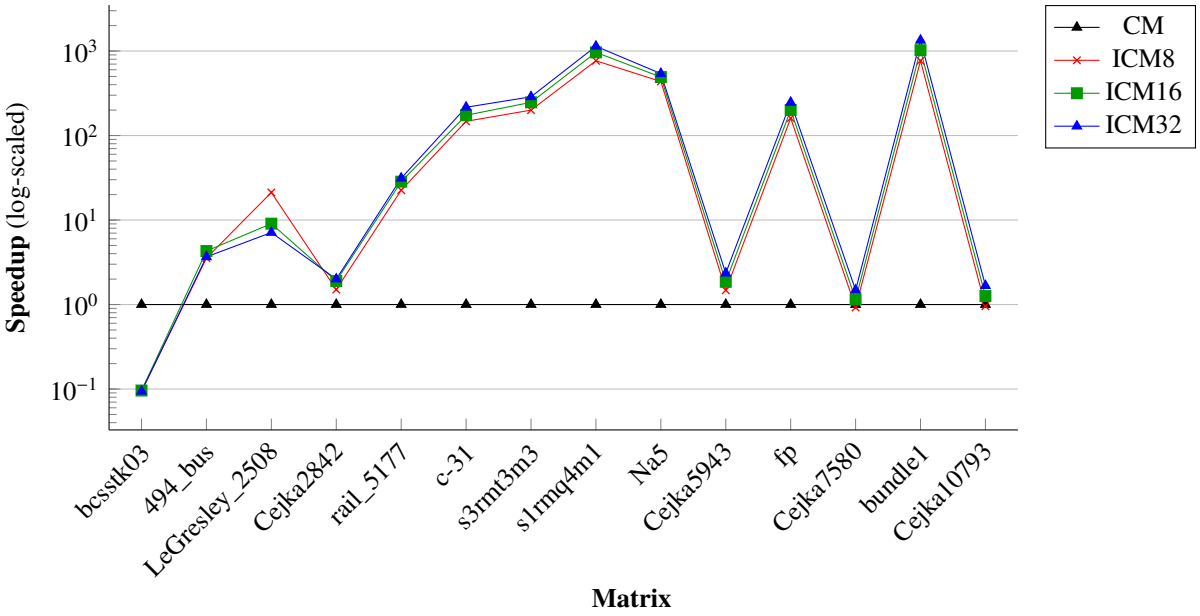
As mentioned in *Implementation requirements* in Section 2.2.1 one of the goals of this project was to "*measure the acceleration of the GPU version of the LU decomposition against the CPU version*". For that reason, speedup from the CM (host) implementation to the ICM*x* (device) implementations will be compared. The comparison - using single and double precision - on the set of matrices listed in Table 3.2 is shown in Figure 3.1 - note that the graphs in the figure have a log-scaled vertical axis for a clearer presentation of differences between the individual implementations.

ICM16 and ICM32 Overall, ICM16 and ICM32 were the best-performing implementations. For single precision, the three highest speedups compared to CM were 2,030.30 by ICM32 and 1,803.08 by ICM16 on the *bundle1* matrix (Figure 3.2a), and 1,352.55 by ICM16 on the *s1rmq4m1* matrix (Figure 3.2b). In terms of double precision, the highest speedups compared to CM were 1,337.53 and 1,137.98 achieved by ICM32 on matrices *bundle1* and *s1rmq4m1* respectively, and 1,022.01 achieved by ICM16 on the *bundle1* matrix.

As can be seen from Figure 3.2b, matrix *s1rmq4m1* has nonzero elements on and near its main diagonal. Thus, the matrices arising from the decomposition ($\mathbb{L}$ and $\mathbb{U}$, or $\mathbb{Z}$) will also have elements mainly around their main diagonals. From the perspective of ICM*x*, this means that the majority of elements requiring computation are found within diagonal sections, therefore, more iterations are required to process them. On the other hand, non-diagonal sections do not require as many iterations as they contain mostly zeros. Consequently, ICM*x* can leverage the fact that each diagonal section has the device's resources available to itself and that non-diagonal sections are processed in fewer iterations which results in a fast decomposition of the input matrix. In other words, ICM*x* is efficient at decomposing n-diagonal sparse matrices, i.e. matrices that have a small number (n) of nonzero diagonals near the main nonzero diagonal. For the *s1rmq4m1* matrix - using either precision - the optimal number of threads per block is either 16×16 or 32×32.



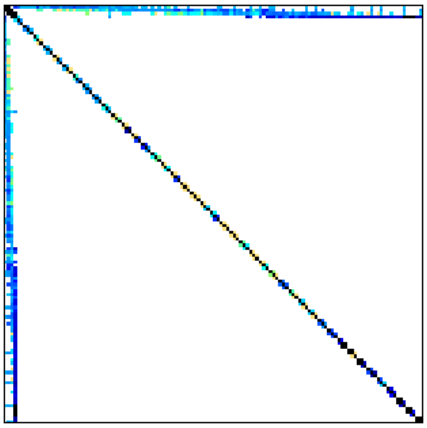
(a) Single precision



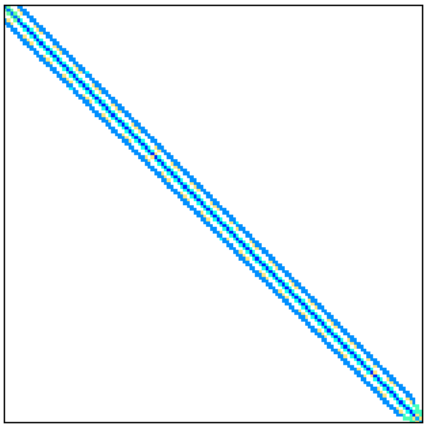
(b) Double precision

Figure 3.1: Speedup comparison between the decomposition times of CM and the ICM*x* implementations on the set of matrices (Table 3.2) using single and double precision. The vertical axis is log-scaled for better visibility of differences between implementations.

Similarly, the main diagonal of the *bundle1* matrix (Figure 3.2a) is also densely populated with nonzero elements. However, it also has nonzero elements in its first few rows and columns. Subsequently, the non-diagonal sections that compute these elements require more iterations in order to be processed. Nevertheless, since the matrix's remaining elements are mostly zero, the other non-diagonal sections require a few iterations - if any - to process their elements. Note that the speedup for *bundle1* is greater



(a) bundle1



(b) s1rmq4m1

Figure 3.2: Nonzero element pattern of the *bundle1* and *s1rmq4m1* matrices. Taken from *The University of Florida Sparse Matrix Collection* [41].

than the speedup for *s1rmq4m1* due to the large difference in matrix dimensions which causes significant problems for CM.

When it comes to suboptimal performance, the speedup of ICM*x* was noticeably low on all dense matrices when using double precision. Since the used dense matrices contain only nonzero elements, every section needed many iterations to be computed which in turn meant that, overall, more iterations were needed to process the entire matrix. For context, the *Cejka5943* dense matrix was decomposed by ICM32 in 44.53 seconds (using double precision), whereas the *Na5* matrix (Figure 3.3) was decomposed in 0.12 seconds.
The reason behind this stark difference in execution times is that the *Na5* matrix has more so an n-diagonal nonzero element structure with minor protrusions in the center. However, the protrusions do not impact the performance severely since the concentration of nonzero elements on the main diagonal is greater than that of the outer diagonals. Therefore, ICM*x* also benefits from the fact that, overall, fewer iterations are required to decompose the entire matrix.

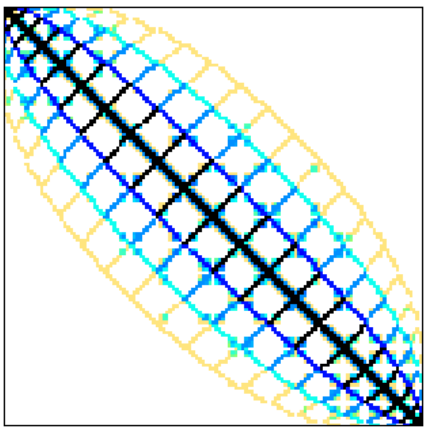
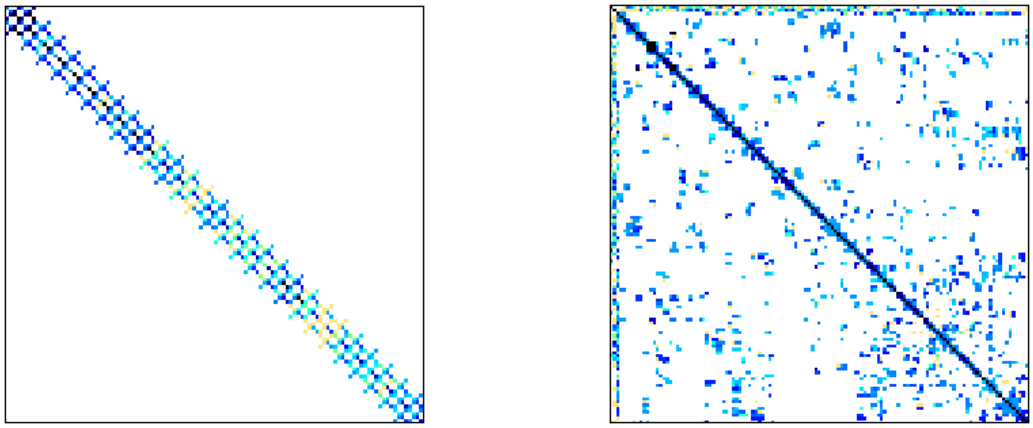


Figure 3.3: Nonzero element pattern of the *Na5* matrix. Taken from *The University of Florida Sparse Matrix Collection* [41].

When single precision was used, the noticeable difference in speedup between sparse and dense matrices was not as prominent.
While the reason behind the significant speedup difference between single and double precision for ICM*x* on dense matrices remains unproven, high-speed shared memory access and generally faster operations using single precision are suspected to be the root causes.

In terms of sparse matrices, ICM*x* achieved higher speedups on all with some exceptions, for example, the *bcsstk03* (Figure **??**) and *rail_5177* (Figure 3.4b) matrices despite the former's nonzero element



(a) bcsstk03 (b) rail_5177

Figure 3.4: Nonzero element pattern of the *bcsstk03* and *rail_5177* matrices. Taken from *The University of Florida Sparse Matrix Collection* [41].

structure being favorable to ICMx. As Table 3.2 lists, the dimensions of the *bcsstk03* matrix are 112×112 which is advantageous for the sequential implementation of CM, as there are fewer steps to perform. Furthermore, since the matrix was already present in host memory, CM's decomposition procedure began without delay. On the other hand, for ICMx, the matrix first had to be copied to the device memory and, additionally, the pre-kernel overhead subroutine of ICMx took - in this case - relatively valuable time to complete. This, combined with the fact that the strength of the iterative algorithm lies in the GPU's ability to process larger datasets, indicates that CM is more suitable for decomposing smaller matrices.

When it comes to the *rail_5177* matrix, irrespective of the fact that its main diagonal is heavily populated with nonzero elements, there are other nonzero elements interspersed in a seemingly random fashion throughout the matrix. Therefore, similarly to dense matrices, the non-diagonal sections of this matrix also require more iterations in order to be processed.

Additionally, when comparing the speedup between single and double precision for ICM16 and ICM32 relative to ICM8, unexpected behavior can be seen for the *LeGresley_2508* matrix (Figure 3.5). When double precision is used, the speedup of ICM16 and ICM32 decreased drastically compared to single precision and compared to the decrease in performance of ICM8. Since $1/10$ of the matrix dimensions are 250.8, `sectionSize` is set to `256` - the edge of the interval. Thus, `sectionSize` is the same for ICM8, ICM16, and ICM32 as their respective `BLOCK_SIZE` is a divisor of `256`. Therefore, from the perspective of software, the only difference between the implementations in this instance - except for the precision - is the different `BLOCK_SIZE`. It can be argued that ICM8 performed better due to the greater granularity of its

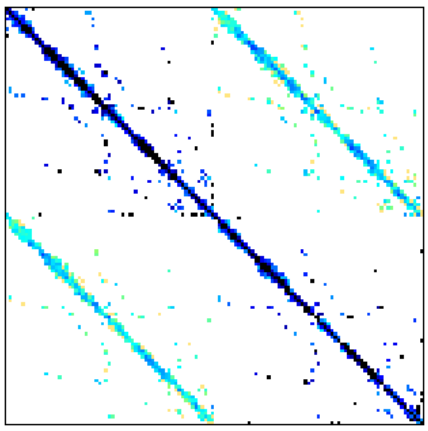


Figure 3.5: Nonzero element pattern of the *LeGresley_2508* matrix. Taken from *The University of Florida Sparse Matrix Collection* [41].

thread blocks combined with the smaller size of the matrix. The greater granularity stems from the fact that ICM8 had 32 blocks assigned to each SM, whereas ICM16 had 8 and ICM32 had 2 - the A100 has a maximum of 2048 threads (64 warps; 32 blocks) per SM [13]. This means that if the GPU would run out of resources in terms of concurrently active threads, then ICM8 would fair better as its smaller blocks could fill up the remaining resources more tightly (detailed in *Grid* in Section 1.2.2).

This effect is only observed when decomposing smaller matrices that have a nonzero element structure similar to that of *LeGresley_2508*, i.e. matrices with nonzero elements on the main diagonal and nonzero elements interspersed elsewhere.

Figure 3.7 seems to further confirm this suspicion for certain matrices where ICM8 outperforms ICM16 and ICM32 when double precision is used. For matrices with larger dimensions, this effect would be mitigated since being able to utilize a few additional blocks would not amount to such a drastic difference in performance. Furthermore, the non-coalesced global memory access present for ICM8 (more so than for ICM16) becomes apparent when decomposing larger matrices - especially when using single precision.

ICM8 In general, ICM8 was faster than CM, however, when its speedup is compared to that of ICM16 and ICM32, there is a difference in performance. In terms of single precision, the speedups achieved seem to be slightly lower for the first 10 matrices (*bcsstk03* to *Cejka5943*), however, they are considerably lower the larger the matrix dimensions become. Specifically, for matrices that require fewer iterations in order to be decomposed, for example, *s1rmq4m1*, *Na5*, and *bundle1*.
Since the execution time for these matrices when using single precision was - at worst - 0.5 seconds for each (0.9 for double precision), loading elements from global to shared memory became a bottleneck. The reason behind this stems from the fact that threads in blocks are divided into warps by their `threadIdx.x` index. Thus, when reading from shared memory, threads in a warp should - ideally - access neighboring global memory addresses. Since elements of the `DenseMatrix` instance are stored in row-major order on the GPU, each warp in ICM8 loads elements from global memory in the following way: the first 8 threads of the warp read data from neighboring addresses, the next 8 threads of the warp read from other neighboring addresses, in other words, they are not found near the first 8 addresses (unless the matrix dimensions are 8×8) - this is non-coalesced access to memory. In this instance, the non-coalesced access is similar to the example depicted in the upper image of Figure 1.12b. Therefore, as mentioned in the figure's caption, instead of the threads in a warp performing one transaction to global memory, they will perform $32/8 = 4$ sequential transactions (detailed in *Global memory* in Section 1.2.3).
This means that non-coalesced access to global memory occurs for ICM8 and ICM16, however, the latter will only perform $32/16 = 2$ sequential transactions. On the other hand, this defect is not present for ICM32 as the threads of a warp access `BLOCK_SIZE = 32` neighboring addresses in global memory. Apart from this behavior being noticeable in Figure 3.1a, it is also apparent for double precision when the vertical axis is not log-scaled, as shown in Figure 3.6 (ICM32 performed better than ICM16 which outperformed ICM8).

CM In Figure 3.1 it can be seen that CM took more time to decompose most of the benchmarked matrices. However, an interesting result arises when comparing CM and ICM8 for double precision. Specifically, for 3 out of the 14 matrices, ICM8 was slower than CM as it achieved the following speedups: 0.09 for *bcsstk03*, 0.92 for *Cejka7580*, and 0.96 for *Cejka10793*. While the result for *bcsstk03* is not surprising given the matrix's dimensions, the others are. The most likely reason for the lower performance is the non-coalesced access to global memory which is most prominent for ICM8.

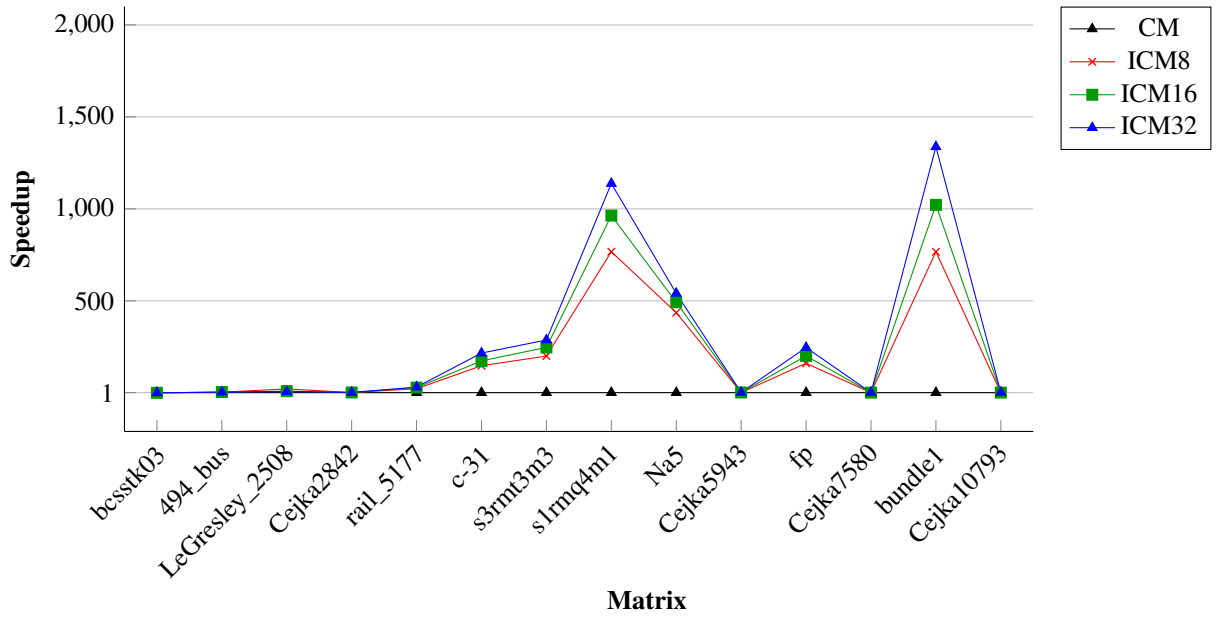


Figure 3.6: Speedup comparison between the decomposition times of CM and the ICMx implementations on the set of matrices (Table 3.2) using **double** precision.

Nevertheless, since the algorithm used for CM is sequential and computes every element of the resulting decomposed matrix $\mathbb{Z}$ regardless of its value, then there is no difference in how it computes matrices with distinct nonzero element structures. Thus, it can be concluded that the performance of CM is determined more so by the dimensions of a matrix, rather than its nonzero element structure.

Overall, the decomposition of matrices using single precision was faster compared to double precision. However, the difference in speed is not clearly visible in either Figure 3.1 (due to the vertical axis being log-scaled) or Figure 3.6 (due to the range of the speedup factor being too great for the values to be distinguishable). For this reason, the raw execution times for single and double precision are included in Table 3.3 and Table 3.4 respectively.

From the results presented in Table 3.3, it can be concluded that ICM32 is the most suitable implementation for decomposing matrices using single precision. In terms of double precision, the same statement could be made when looking at Table 3.4. However, since the results presented in the mentioned tables were only a small subset of all results, verification of this claim is required for the entire set of 63 matrices.

3.4.2 Performance of Implementations Across All Matrices

This section shows the speedup comparison and accuracy of results using both single and double precision for the decomposition implementations listed in Section 3.1 across the entire set of 63 matrices. The benchmarks were run on the platform specified in Table 3.1. To illustrate how ICMx compares to CM on matrices that increase in size, the matrices in the graphs were sorted from smallest to largest. The speedup comparison between CM and ICMx can be found in Figure 3.7 and the accuracy of the results can be found in Figure 3.8. All graphs in this section have a log-scaled vertical axis as the range of values was too great to present any coherent data.

Matrix	CM	ICM8	ICM16	ICM32
bcsstk03	0.0002	0.0022	0.0020	0.0022
494_bus	0.0292	0.0049	0.0053	0.0043
LeGresley_2508	4.2193	0.0793	0.0498	0.0531
Cejka2842	6.2210	3.0391	2.1182	2.1412
rail_5177	55.1124	1.4350	1.0669	1.1022
c-31	59.3208	0.1598	0.1330	0.1327
s3rmt3m3	61.9867	0.1450	0.1133	0.1684
s1rmq4m1	67.8509	0.0641	0.0502	0.0505
Na5	62.3995	0.0817	0.0636	0.0649
Cejka5943	90.1114	0.7035	0.5230	0.4856
fp	179.7054	0.5278	0.3980	0.3690
Cejka7580	151.7981	12.2164	3.0289	2.2853
bundle1	617.5120	0.4853	0.3425	0.3041
Cejka10793	643.3462	9.2837	2.8510	2.2107

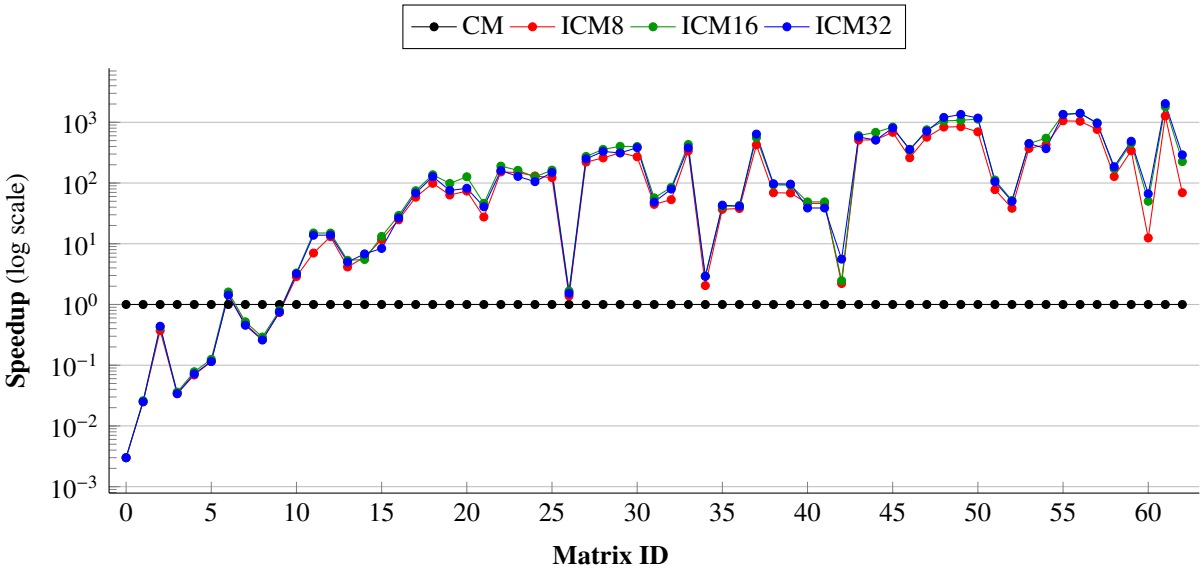
Table 3.3: The execution time (in seconds) of decomposition for all implementations on the set of matrices (Table 3.2) using **single** precision. The fastest time for each matrix is highlighted in green.

Matrix	CM	ICM8	ICM16	ICM32
bcsstk03	0.0003	0.0027	0.0026	0.0027
494_bus	0.0299	0.0084	0.0070	0.0082
LeGresley_2508	4.3542	0.2055	0.4805	0.6118
Cejka2842	6.9110	4.5995	3.6436	3.4559
rail_5177	67.9030	3.0109	2.3882	2.1754
c-31	72.3863	0.4913	0.4166	0.3349
s3rmt3m3	74.0106	0.3704	0.2997	0.2577
s1rmq4m1	81.2749	0.1059	0.0843	0.0714
Na5	67.2166	0.1543	0.1365	0.1243
Cejka5943	104.6150	70.8865	56.2768	44.5275
fp	154.8232	0.9641	0.7715	0.6287
Cejka7580	159.6421	172.9549	138.9242	107.8041
bundle1	679.3846	0.8859	0.6648	0.5079
Cejka10793	699.9634	731.9412	556.0243	419.1401

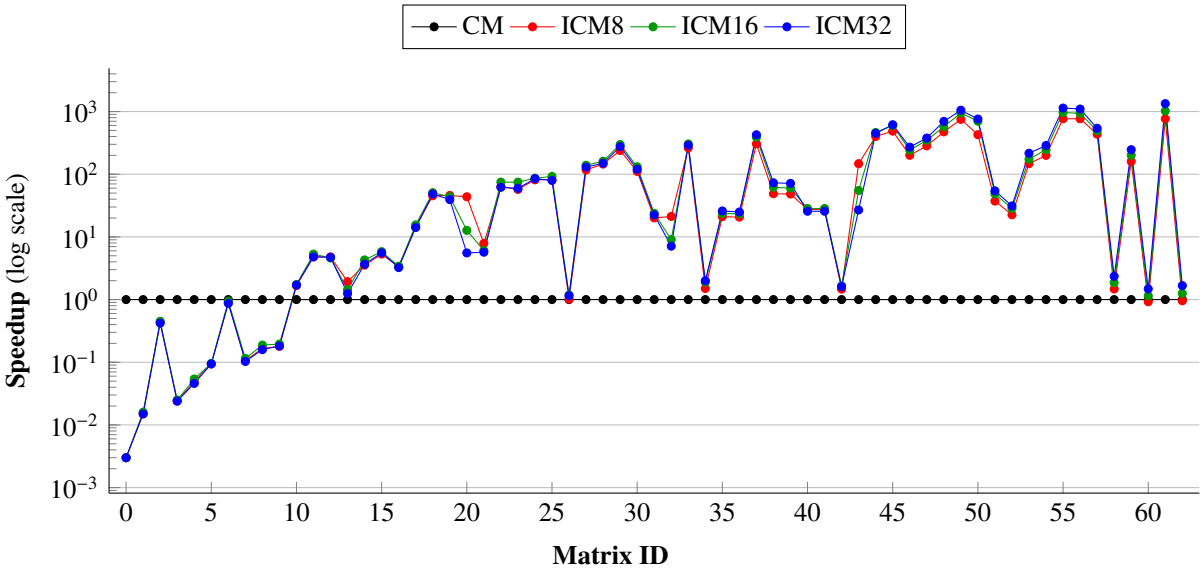
Table 3.4: The execution time (in seconds) of decomposition for all implementations on the set of matrices (Table 3.2) using **double** precision. The fastest time for each matrix is highlighted in green.

Speedup As shown in Figure 3.7, in terms of speedup compared to CM, the ICMx implementations achieved similar results.

Specifically, when single precision was used, all ICMx implementations were faster in 54/63 cases. In terms of double precision, ICM32 and ICM16 were faster in 53/63, and ICM8 was faster in 51/63 cases. In other words, contrary to the results of decomposition performance on the subset of 14 matrices presented in Section 3.4.1, these results show that CM outperformed ICMx in more cases than just one. Although, it must be mentioned that the matrices on which CM performed better than ICMx did have dimensions between 27×27 and 415×415 (with the two exceptions for ICM8 detailed in *CM* in Section 3.4.1).



(a) Single precision



(b) Double precision

Figure 3.7: Speedup comparison between CM and ICMx implementations on the entire set of 63 matrices using both single and double precision. Matrix ID signifies the ID of the matrices after they have been sorted according to their dimension from smallest to largest. The vertical axis is log-scaled for better visibility of differences between implementations.

When it comes to single precision, disregarding the fact that all ICMx implementations were faster in the same number of cases, a lack of performance of ICM8 can be seen. It appears that the issue with non-coalesced access to global memory made a difference. This is further supported by the fact that after a certain point (matrix 46) ICM8 appears to be consistently falling behind ICM32 - especially noticeable for the larger dense matrices when using single precision (the three dips in speedup after matrix 55 in Figure 3.7a).

In terms of double precision, the performance of ICMx appears more consistent. The general rule of ICM8 performing worse than ICM16 and ICM32 for larger matrices still stands and the performance of ICM16 for matrices past matrix 30 is - in the majority of cases - less than that of ICM32. The dips in the speedup of ICMx for matrices past matrix 25 are caused by the dense matrices found in the set.

Additionally, Table 3.5 shows the total time required by each implementation to decompose the set of 63 matrices using both single and double precision. The data in this table provides further support to the claim made at the end of Section 3.4.1 which stated that ICM32 is the most suitable implementation for single and double precision.

Precision	CM	ICM8	ICM16	ICM32
Single	2,393.09	35.88	17.36	13.84
Double	2,635.45	1,000.27	772.29	592.19

Table 3.5: The total execution time (in seconds) taken by each implementation to decompose the set of 63 matrices (Table 3.2) on the RCI compute cluster specified in Table 3.1 using single and double precision. The fastest time for each precision is highlighted in green.

Accuracy of results While this part is the other key factor of the implementation (the first being speed of execution), it serves more as a means to verify that the presented performance does not come at the cost of drastically inaccurate results. Figure 3.8 shows the maximum difference between the input matrix $\mathbb{A}$ and the multiplication of the decomposed matrices produced by the implementations ($\mathbb{L}$ and $\mathbb{U}$, or $\mathbb{Z}$), i.e.

$$\max |\mathbb{A} - \mathbb{L}\mathbb{U}| \ .$$

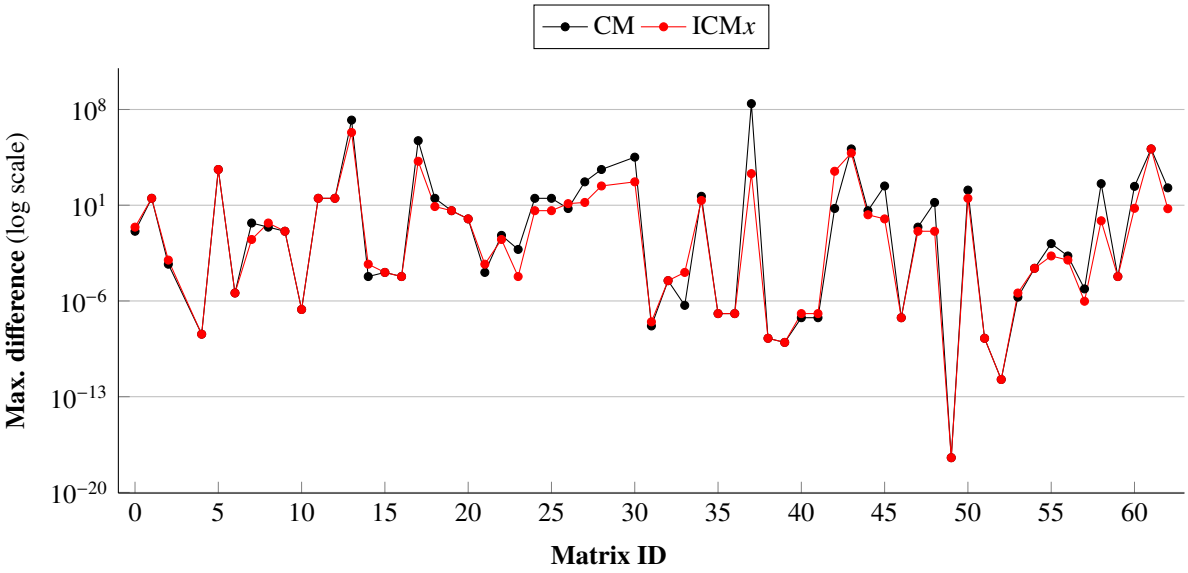
In other words, the matrix that results from multiplying the matrices obtained from the decomposition ($\mathbb{L}$ and $\mathbb{U}$) should - ideally - be the same as the input matrix ($\mathbb{A}$). The values presented in this part are the largest differences between the actual results and the ideal results.

For this project, the maximum difference will also be referred to as the *error*. Similarly to the previous figures, the vertical axis is log-scaled to present the results more clearly. The graphs in the figures only show data for ICMx as the accuracy of the results did not differ for the ICMx implementations.

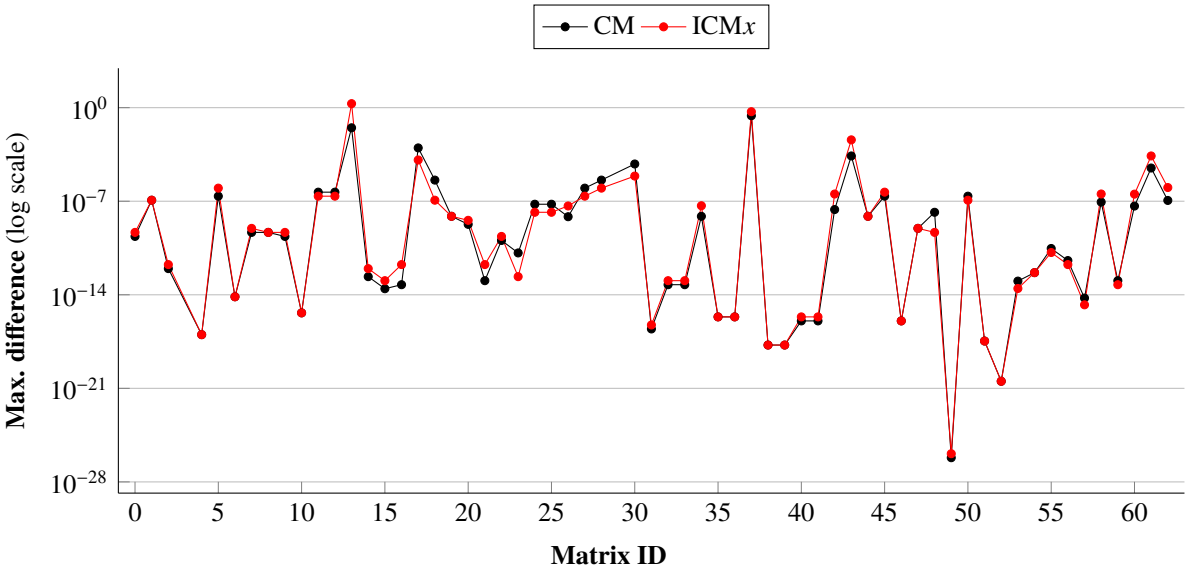
Overall, it seems that the accuracy of results produced by both approaches (CM and ICMx) is similar, nevertheless, values for both single and double precision will be analyzed.

Single precision The statistical analysis of the accuracy of results for single precision is presented using a collection of basic statistical indexes in Table 3.6.

The values from the table present the problem of using single precision. Even though CM is a sequential algorithm and should provide the exact solution the results produced by it can be drastically different from those expected. Specifically, the average error for CM was 4,540,350.594 which is not usable. On the other hand, since the median is 0.125, and the smallest difference is 0 it seems that the large average was caused by a few extreme values. This claim is further supported by the maximum error achieved by CM which is 268,435,500.



(a) Single precision



(b) Double precision

Figure 3.8: The accuracy of results achieved by CM and ICMx implementations on the entire set of 63 matrices using both single and double precision. In this instance, the term "accuracy" refers to the maximum difference of $|\mathbb{A} - \mathbb{L}\mathbb{U}|$ where $\mathbb{A}$ is the input matrix and $\mathbb{L}$ and $\mathbb{U}$ are the matrices obtained from the decomposition procedure. The higher the maximum difference, the lower the accuracy of the results (the higher the error). Matrix ID signifies the ID of the matrices after they have been sorted according to their dimension from smallest to largest. The vertical axis is log-scaled for better visibility of differences between implementations. Note that the vertical axes of both graphs do not cover the same range.

For ICMx, the average error was 36,830.43 which is significantly lower than that of CM. Furthermore, the maximum error and median value of errors achieved by ICMx are all lower than that of CM. Additionally, the standard deviation is 128 times smaller than that of CM.

Accuracy index	CM	ICMx
Average	4,540,350.594	36,830.430
Maximum	268,435,500.000	2,097,152.000
Minimum	0.000	0.000
Median	0.125	0.031
Std. dev.	33,580,198.190	262,296.394

Table 3.6: Statistical indexes for the accuracy of the results obtained from the decomposition procedure performed by CM and ICMx using single precision. The accuracy of the results was rounded to three decimal places as that was sufficient to portray the characteristics of the dataset.

In summary, all of the facts mentioned for the accuracy of results when using single precision can be used to further promote the claim made at the end of *Numerical Method* in Section 1.3.2: "*Since Crout's method is direct, it can be theorized that rounding errors may result in it providing less accurate results compared to its numerical modification*". In other words, for single precision, the results suggest that ICMx is more accurate than CM. However, since the set contained only 63 matrices, further benchmarks on larger sets are required to verify this claim.

Double precision Similarly to single precision, the statistical analysis of the accuracy of results for double precision is presented using a collection of basic statistical indexes in Table 3.7.

Accuracy index	CM	ICMx
Average	0.0045	0.0398
Maximum	0.2500	2.0000
Minimum	0.0000	0.0000
Median	2.3283×10^{-10}	4.6566×10^{-10}
Std. dev.	0.0314	0.2566

Table 3.7: Statistical indexes for the accuracy of the results obtained from the decomposition procedure performed by CM and ICMx using double precision. The accuracy of the results was rounded to four decimal places to portray the characteristics of the dataset. Furthermore, some values were simply too small and as such, they were listed using scientific notation.

Unlike the accuracy of results for single precision, the results achieved using double precision were less erroneous. The average error in the results produced by CM was 0.0045, the maximum error was 0.25 and the minimum error was 0. Furthermore, the median error was at a very accurate 2.33×10^{-10}. However, the most drastic difference when comparing the errors of CM between single and double precision is the standard deviation. For the former, the deviation of errors was 33,580,198.19, whereas for the latter it was 0.0314.

When comparing CM to ICMx for double precision, it can be seen that the roles have switched and that ICMx is now the less-accurate implementation across all indexes. For ICMx, the average error is 8 times larger than that of CM and its median is double that of CM. Furthermore, the standard deviation of errors for ICMx is also 8 times greater than that of CM.

Ultimately, it can be concluded that CM is the format that produces more accurate matrices when using double precision. However, the performance of CM remains to be multitudes lower than that of ICMx. Further benchmarks on real-life problems are required to determine whether the errors achieved by ICMx are too severe, or if the speedup compared to CM is worth the slightly more inaccurate results.

3.4.3 Comparison of Optimizations

This section aims to show the evolution of the ICMx implementations throughout the development process. Since the benchmarks for each optimization were executed during the development process, they were run on a different platform than the results presented in Sections 3.4.1 and 3.4.2. The platform used during development was the GP7 compute server (specifications listed in Table 3.8) which is located in the Faculty of Nuclear Sciences and Physical Engineering of the Czech Technical University in Prague.

CPU	Intel(R) Core(TM) i9-9900KF @ 3.60GHz (8 cores, 16 threads)
RAM	62GB RAM
GPU	2× Nvidia GeForce RTX 3060 12GB GDDR6 (360 GByte/s mem. bandwidth)
Operating System	Arch Linux (Kernel Version: 5.18.8)
Compiler	GCC 12.1.1
CUDA	CUDA 11.7

Table 3.8: Specifications of GP7 compute server that the benchmarks were run on during development.

Furthermore, a subset of the set of 63 matrices was used for measuring the performance of implementations during development. The reasons for this were the following:

- The initial implementations on the GPU were egregiously slow, thus, only sparse matrices smaller than $6,000 \times 6,000$ were used at the time.
- The final set of 63 matrices underwent numerous changes as newly-found matrices were added throughout the development process. Only once the GPU implementation became adequately performant, larger and dense matrices were added. Once the set was completed, there was an attempt to redo the benchmark results for key optimizations, however, the compute cluster time limit was not sufficient as the decomposition of some dense matrices would require up to two days with the initial GPU implementations.

The set of matrices that was used to measure performance during development comprised 50 sparse matrices with dimensions ranging from 27×27 to $5,832 \times 5,832$.

Other than the key optimizations mentioned in Section 2.3, two additional will be SMopt to show the effect that the value of `sectionSize` can have on both the speed of the computation and the accuracy of the results. Specifically, the key optimizations that will be presented are the following:

- **Base** - The naive implementation (mentioned in *Initial naive implementation* in Section 2.3).

- **SM** - The first version of the implementation that used shared memory (mentioned in *Calculation by tiles using shared memory* in Section 2.3).
- **SMopt** - The optimized version of the shared-memory implementation (mentioned in *Elimination of conditions* in Section 2.3).
- **ProcRow5** - The initial implementation of *Processing by row sections* where each section consisted of 512 rows with *tol* = 0.005.
- **ProcRow0** - The upgraded version of *Processing by row sections* where *tol* = 0.
- **ParSecGPU** - The version of *Processing by parallel sections* where `sectionSize` was limited by the capabilities of the GPU (mentioned in *Section size dependent on maximum active threads* in Section 2.3).
- **ICM32** - The final version of the implementation (mentioned in *Iterative Crout method* in Section 2.2.1).

Note that the tolerance value *tol* was set to 0.001 for the *Base* optimization and 0.005 for the *SM*, and *SMopt*.

Execution times The results will be presented only for double precision and for `BLOCK_SIZE` set to 32. The reason behind this is that the results are shown in order to detail the evolution of the implementation, rather than compare differences in thread-per-block configurations. The execution times for the decomposition of the set of 50 matrices for the optimizations listed above are shown in Figure 3.9 (the vertical axis is log-scaled for clarity) and the total time taken for each optimization to decompose the set of 50 matrices is shown in Table 3.9.

As can be seen from Figure 3.9 the *Base* optimization (orange line) is found above the other optimizations for the majority of matrices, which indicates that it took the most time to decompose each matrix. This fact is further supported by the total time taken by *Base* to decompose the entire set of 50 matrices, which is shown in Table 3.9. According to the graph, it would seem that the *SM* and *SMopt* optimizations achieved similar times, however, as can be seen in Table 3.9, they were almost 3 and 4 times faster, respectively, than *Base*.

Optimization	Base	SM	SMopt	ProcRow5	ProcRow0	ParSecGPU	ICM32
Time [s]	2,140.83	745.15	514.91	57.79	143.68	89.91	43.88

Table 3.9: The total execution time (in seconds) taken by each optimization to decompose the set of 50 matrices.

It may be difficult to discern the performance of the remaining optimizations in Figure 3.9. However, the results in Table 3.9 show that ICM32 was the fastest overall. In general, the table effectively shows how each optimization helped improve the speed of decomposition - specifically the total execution time decreased almost 50 times from *Base* to ICM32. For reference, the total execution time required for ICM32 to decompose this subset of 50 matrices on the RCI compute cluster node (presented in Table 3.1) was 5.02 seconds.
It can be seen that the total execution time for *ProcRow5* was 57.79 seconds, which is significantly

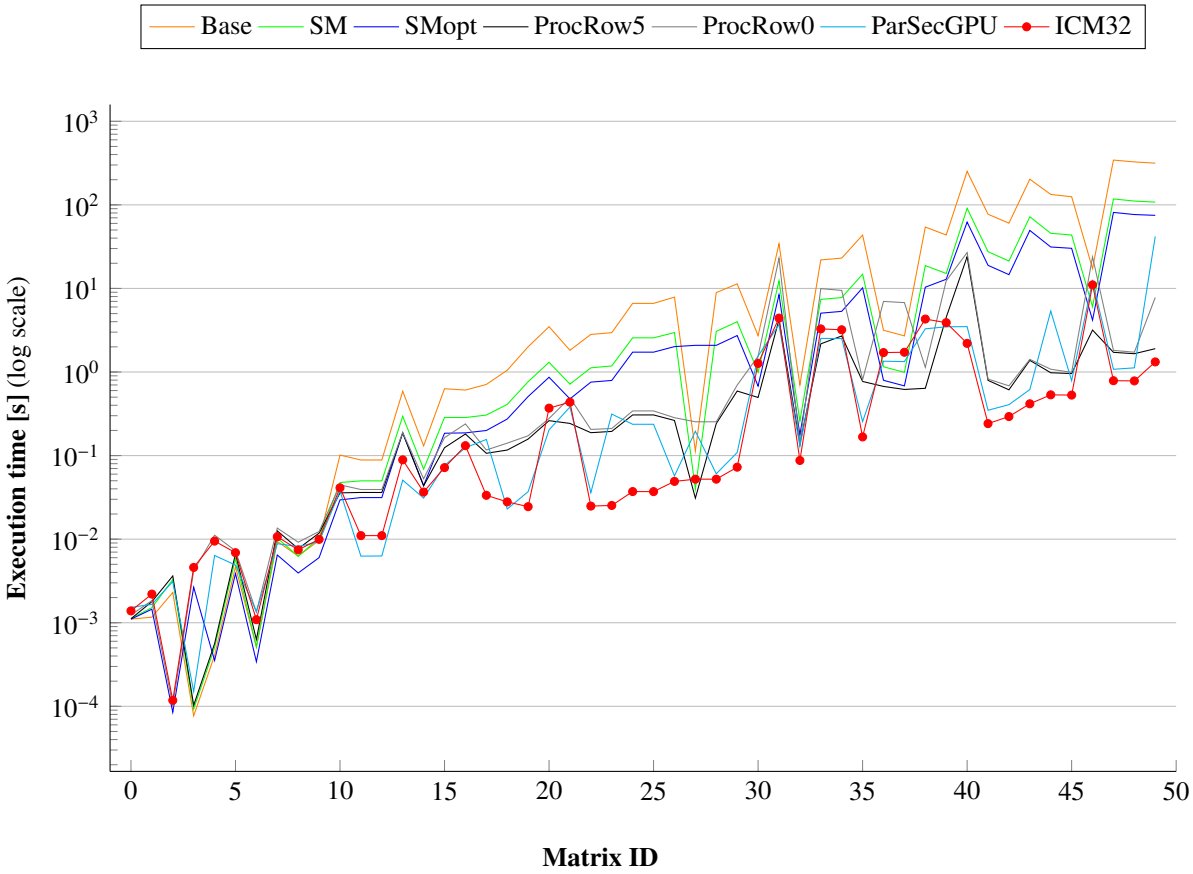


Figure 3.9: The execution times (in seconds) taken by each decomposition optimization listed above to decompose the set of 50 matrices using double precision and with `BLOCK_SIZE` set to 32. Matrix ID signifies the ID of the matrices after they have been sorted according to their dimension from smallest to largest. The vertical axis is log-scaled for better visibility of differences between optimizations.

faster than the other optimizations (apart from ICM32). The reason behind this is that *ProcRow5* had its tolerance value *tol* set to 0.005, thus, as can be seen in Table 3.10 the accuracy of the results it produced from the set of 50 matrices was not as good as that of ICM32.

Accuracy In terms of accuracy, Table 3.10 shows that *ProcRow0* achieved the most accurate decomposition results with an average error of 0.0397 and a standard deviation of 0.1752. This is due to the fact that the optimization processed the input matrix by sections comprised of 512 rows. This means that if the tolerance value *tol* was set to zero, then the optimization would perform many iterations. However, this gain in accuracy is reflected in the relatively poor performance - shown in Table 3.9. Specifically, *ProcRow0* required 143.68 seconds to decompose all 50 matrices.

When both speed and accuracy were taken into account, the ICM32 optimization excelled more so in speed, than in accuracy. However, as mentioned before, since the execution time of the most-accurate optimization (*ProcRow0*) was insufficient the procedure used in ICM32 was selected as the optimal choice out of all the optimizations.

Accuracy index	Base	ProcRow5	ProcRow0	ParSecGPU	ICM32
Average	0.8776	0.1559	0.0397	0.7004	0.0595
Maximum	11.0353	5.6900	1.0000	29.7813	2.0000
Minimum	0.0001	0.0000	0.0000	0.0000	0.0000
Median	0.0722	8.57963×10^{-5}	1.25712×10^{-10}	4.07454×10^{-10}	1.25712×10^{-10}
Std. dev.	1.8660	0.8097	0.1752	4.1742	0.2979

Table 3.10: Statistical indexes for the accuracy of the results obtained from the decomposition procedure performed by the optimizations listed at the beginning of this section. The procedure was done using double precision and with `BLOCK_SIZE` set to 32.

Conclusion

The objective of this project was to study, implement, and compare the performance of the GPU implementation of LU decomposition to the CPU implementation.

First, the comparison of recent CPUs and GPUs was presented in order to lay a foundation of the advantages and disadvantages of using either. Then, the nuances of the software layer used to orchestrate the execution of code on the GPU were presented - from CUDA's thread and memory management systems to its sophisticated concurrent execution solution and examples that tied together the theory presented. Furthermore, the LU decomposition method was shown in both its direct and iterative versions.

Following the theory, the TNL project - which provided a data structure paramount to the development - was briefly described. Taking the theory and the dependencies into account, subsequently, the project that contained the implementation of the LU decomposition in its CPU and GPU form was detailed. In addition to the methods' implementations, unit tests assuring their quality and benchmarks measuring their performance were added to the project. Furthermore, the evolution of the GPU's LU decomposition implementation was expounded.

Finally, the speed and accuracy of the CPU and GPU implementations were compared using the benchmark structure incorporated in the project. Specifically, the process of measuring their performance consisted of decomposing a curated set of 63 matrices with varying characteristics on a state-of-the-art compute cluster. While the results of the benchmarks indicated that the GPU version of LU decomposition with 32 threads per block was the optimal choice in general, other implementations were, in specific cases, found to be more suitable. Overall, the performance of the CPU version was suboptimal compared to that of the GPU version, with the following exceptions: decomposing matrices smaller than 500×500 and dense matrices using double precision - the latter instance of outperformance was only present when the CPU version was compared to the 8-thread-per-block GPU version.

In terms of future work, as mentioned in *Benchmark Results* in Section 3.4 the benchmarks were run on a set of only 63 matrices. While this number is sufficient for a preliminary comparison of implementations it is not enough to make definitive claims on their overall performance. Furthermore, notwithstanding the promising results, it is necessary to compare the performance of both the CPU and GPU versions to a well-established solution on real-life problems. Moreover, in addition to detailing the evolution of the GPU implementation, Section 2.3 *Optimization* mentions possible improvements that may greatly increase its performance. In summary, future work includes both the extension of benchmarks and further optimizations of the implementations.

Bibliography

[1] ČEJKA, L. *Formats for storage of sparse matrices on GPU*. Prague, 2020. Bachelor's Degree Project. Czech Technical University in Prague.

[2] NVIDIA, C. *CUDA C++ Programming Guide* [online]. [visited on 2022-05-19]. Available from: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[3] GLAWION, A. *Server vs. Desktop CPUs: What are the differences?* [online]. [visited on 2022-05-19]. Available from: https://www.cgdirector.com/server-vs-desktop-cpus/.

[4] HAGEDOORN, H. *Nvidia GeForce GTX 1070 review - Pascal GPU Architecture* [online]. [visited on 2022-05-22]. Available from: https://www.guru3d.com/articles-pages/nvidia-geforce-gtx-1070-review,3.html.

[5] NVIDIA. *NVIDIA TURING GPU ARCHITECTURE: Graphics Reinvented* [online]. [visited on 2022-05-22]. Available from: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[6] SMITH, R. *NVIDIA Posts Full GeForce GTX 1070 Specifications: 1920 CUDA Cores Boosting to 1.68GHz* [online]. [visited on 2022-05-22]. Available from: https://www.anandtech.com/show/10336/nvidia-posts-full-geforce-gtx-1070-specs.

[7] TECHPOWERUP. *NVIDIA GeForce GTX 1070* [online]. [visited on 2022-05-22]. Available from: https://www.techpowerup.com/gpu-specs/geforce-gtx-1070.c2840.

[8] WALTON, J. *Nvidia GeForce RTX 3060 12GB Review: Hope Springs Eternal* [online]. [visited on 2022-05-22]. Available from: https://www.tomshardware.com/reviews/nvidia-geforce-rtx-3060-review.

[9] W1ZZARD. *MSI GeForce RTX 3060 Gaming X Trio Review: The GeForce Ampere Architecture* [online]. [visited on 2022-05-22]. Available from: https://www.techpowerup.com/review/msi-geforce-rtx-3060-gaming-x-trio/2.html.

[10] TECHPOWERUP. *NVIDIA GeForce RTX 3060* [online]. [visited on 2022-05-22]. Available from: https://www.techpowerup.com/gpu-specs/geforce-rtx-3060.c3682.

[11] MAY, K. *NVIDIA GeForce RTX 3060 Ti Founders Edition Graphics Card Review* [online]. [visited on 2022-05-22]. Available from: https://wccftech.com/review/nvidia-geforce-rtx-3060-ti-founders-edition-graphics-card-review/2/.

[12] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE: THE WORLD'S MOST ADVANCED DATA CENTER GPU* [online]. [visited on 2022-08-25]. Available from: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[13] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture: UNPRECEDENTED ACCELERATION AT EVERY SCALE* [online]. [visited on 2022-05-22]. Available from: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[14] NVIDIA. *NVIDIA A100 TENSOR CORE GPU: Unprecedented acceleration at every scale* [online]. [visited on 2022-05-22]. Available from: https://www.nvidia.com/en-us/data-center/a100/.

[15] OH, F. *What Is CUDA?* [online]. [visited on 2022-05-20]. Available from: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.

[16] DOGMA1138. *CUDA support much more languages than just C++ and Fortran* [online]. [visited on 2022-05-22]. Available from: https://news.ycombinator.com/item?id=26605219.

[17] RUETSCH, G.; OSTER, B. *Getting Started with CUDA* [online]. [visited on 2022-06-07]. Available from: https://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf.

[18] MARZIALE, L.; MOVVA, S.; III, G. G. R.; ROUSSEV, V.; SCHWIEBERT, L. Massively Threaded Digital Forensics Tools. In: *Handbook of Research on Computational Forensics, Digital Crime, and Investigation* [online]. IGI Global, 2010, pp. 234–256 [visited on 2022-05-30]. ISBN 9781605668369. Available from DOI: 10.4018/978-1-60566-836-9.ch010.

[19] DURANT, L.; GIROUX, O.; HARRIS, M.; STAM, N. *Inside Volta: The World's Most Advanced Data Center GPU* [online]. [visited on 2022-05-30]. Available from: https://developer.nvidia.com/blog/inside-volta/.

[20] ABI-CHAHLA, F. *Nvidia's CUDA: The End of the CPU?* [online]. [visited on 2022-06-05]. Available from: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html.

[21] HSIAO, Y. *GPU: CUDA intro* [online]. [visited on 2022-06-11]. Available from: https://hackmd.io/@yaohsiaopid/ryHNKkxTr?type=view.

[22] ROSE, C. *Cuda Succinctly*. 1st edition. United States: CreateSpace Independent Publishing Platform, 2017. ISBN 9781542827409.

[23] HARRIS, M. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels* [online]. [visited on 2022-06-14]. Available from: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.

[24] CABRERA, F. *The CUDA Parallel Programming Model - 5. Memory Coalescing* [online]. [visited on 2022-06-14]. Available from: https://nichijou.co/cuda5-coalesce/.

[25] MARTÍNEZ, M. U. *CUDA Optimizations, Debugging and Profiling* [online]. [visited on 2022-06-14]. Available from: http://materials.prace-ri.eu/35/1/gpuvideo4.pdf.

[26] HERNÁNDEZ, M.; GUERRERO, G. D.; CECILIA, J. M.; GARCÍA, J. M.; INUGGI, A.; JBABDI, S.; BEHRENS, T. E. J.; SOTIROPOULOS, S. N.; YACOUB, E. Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs. *PLoS ONE* [online]. 2013-4-29, vol. 8, no. 4, p. 2 [visited on 2022-05-20]. ISSN 1932-6203. Available from DOI: 10.1371/journal.pone.0061892.

[27] RENNICH, S.; NVIDIA. *CUDA C/C++ Streams and Concurrency* [online]. [visited on 2022-06-07]. Available from: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[28] NVIDIA. *CUDA Runtime API: API Reference Manual* [online]. [visited on 2022-06-21]. Available from: https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf.

[29] MCKENNON, J. *CUDA Parallel Thread Management* [online]. [visited on 2022-06-23]. Available from: https://www.microway.com/hpc-tech-tips/cuda-parallel-thread-management/.

[30] HARRIS, M. *Using Shared Memory in CUDA C/C++* [online]. [visited on 2022-06-25]. Available from: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[31] LINDFIELD, G.; PENNY, J. Linear Equations and Eigensystems. In: *Numerical Methods* [online]. Elsevier, 2019, pp. 73–156 [visited on 2022-06-28]. ISBN 9780128122563. Available from DOI: 10.1016/B978-0-12-812256-3.00011-7.

[32] PRESS, W. H. *Numerical recipes: the art of scientific computing*. 3rd ed. Cambridge: Cambridge University Press, 2007. ISBN 9780521880688.

[33] VISMOR. *4.3 Crout's LU Factorization* [online]. [visited on 2022-06-28]. Available from: https://vismor.com/documents/network_analysis/matrix_algorithms/S4.SS3.php.

[34] *LU decomposition* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [visited on 2022-06-28]. Available from: https://en.wikipedia.org/wiki/LU_decomposition.

[35] ANZT, H.; RIBIZEL, T.; FLEGAR, G.; CHOW, E.; DONGARRA, J. ParILUT - A Parallel Threshold ILU for GPUs. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* [online]. IEEE, 2019, pp. 231–241 [visited on 2022-05-03]. ISBN 978-1-7281-1246-6. Available from DOI: 10.1109/IPDPS.2019.00033.

[36] *Template Numerical Library* [online]. [visited on 2022-07-30]. Available from: https://tnl-project.gitlab.io/tnl/.

[37] OBERHUBER, T.; KLINKOVSKÝ, J.; FUČÍK, R. TNL: NUMERICAL LIBRARY FOR MODERN PARALLEL ARCHITECTURES. *Acta Polytechnica* [online]. 2021-02-10, vol. 61, no. SI, pp. 122–134 [visited on 2022-07-30]. ISSN 1805-2363. Available from DOI: 10.14311/AP.2021.61.0122.

[38] CHOW, E.; PATEL, A. Fine-Grained Parallel Incomplete LU Factorization. *SIAM Journal on Scientific Computing* [online]. 2015, vol. 37, no. 2, pp. C169–C193 [visited on 2022-08-10]. ISSN 1064-8275. Available from DOI: 10.1137/140968896.

[39] CARDOSO, J. M.; COUTINHO, J. G. F.; DINIZ, P. C. Source code transformations and optimizations. In: *Embedded Computing for High Performance* [online]. Elsevier, 2017, pp. 137–183 [visited on 2022-08-13]. ISBN 9780128041895. Available from DOI: 10.1016/B978-0-12-804189-5.00005-3.

[40] *RCI Cluster Hardware* [online]. [visited on 2022-08-16]. Available from: https://login.rci.cvut.cz/wiki/hardware.

[41] DAVIS, T. A.; HU, Y. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software* [online]. 2011, vol. 38, no. 1, pp. 1–25 [visited on 2022-08-16]. ISSN 0098-3500. Available from DOI: 10.1145/2049662.2049663.

Appendix A

Crout Method Implementation for the CPU

The Crout method implementation for the CPU using matrices $\mathbb{L}$ and $\mathbb{U}$ is shown in Listing A.1 and using matrix $\mathbb{Z}$ in Listing A.2.

```
1  template< typename Matrix1, typename Matrix2, typename Matrix3 >
2  void CroutMethod::decompose( Matrix1& A, Matrix2& L, Matrix3& U )
3  {
4    using RealType  = typename Matrix1::RealType;
5    using IndexType = typename Matrix1::IndexType;
6
7    IndexType num_rows = A.getRows();
8    IndexType num_cols = A.getColumns();
9
10   TNL_ASSERT_EQ( num_rows, num_cols, "Matrix A must be a square matrix!" );
11
12   IndexType i, j, k;
13   RealType sum = 0;
14
15   for( j = 0; j < num_rows; ++j ) {
16     for( i = j; i < num_rows; ++i ) {
17       sum = 0;
18       for( k = 0; k < j; ++k ) {
19         sum = sum + L.getElement( i, k ) * U.getElement( k, j );
20       }
21
22       L.setElement( i, j, A.getElement( i, j ) - sum );
23     }
24
25     for( i = j; i < num_rows; ++i ) {
26       TNL_ASSERT( L.getElement( j, j ) != 0, std::cerr << "L( " << i << ", " << ↩
             j << " ) = 0. Cannot divide by 0." << std::endl );
27       sum = 0;
28       for( k = 0; k < j; ++k ) {
29         sum = sum + L.getElement( j, k ) * U.getElement( k, i );
30       }
31
32       U.setElement( j, i, ( A.getElement( j, i ) - sum ) / L.getElement( j, j ) ↩
             );
33     }
```

```
34      }
35   }
```

Listing A.1: Implementation of the Crout method on the CPU using matrices $\mathbb{L}$ and $\mathbb{U}$. All matrix and variable types are obtained from template arguments of the method. Taken from the Decomposition project repository on GitLab[4].

```
1   template< typename Matrix1, typename Matrix2 >
2   void CroutMethod::decompose( Matrix1& A, Matrix2& Z )
3   {
4     using RealType  = typename Matrix1::RealType;
5     using IndexType = typename Matrix1::IndexType;
6
7     IndexType num_rows = A.getRows();
8     IndexType num_cols = A.getColumns();
9
10    TNL_ASSERT_EQ( num_rows, num_cols, "Matrix A must be a square matrix!" );
11
12    IndexType i, j, k;
13    RealType sum = 0;
14
15    for( j = 0; j < num_rows; ++j ) {
16      for( i = j; i < num_rows; ++i ) {
17        sum = 0;
18        for( k = 0; k < j; ++k ) {
19          sum = sum + Z.getElement( i, k ) * Z.getElement( k, j );
20        }
21
22        Z.setElement( i, j, A.getElement( i, j ) - sum );
23      }
24
25      for( i = j; i < num_rows; ++i ) {
26        if( j == i ) continue;
27
28        TNL_ASSERT( Z.getElement( j, j ) != 0, std::cerr << "Z( " << j << ", " << ↩
              j << " ) = 0. Cannot divide by 0." << std::endl );
29        sum = 0;
30        for( k = 0; k < j; ++k ) {
31          sum = sum + Z.getElement( j, k ) * Z.getElement( k, i );
32        }
33
34        Z.setElement( j, i, ( A.getElement( j, i ) - sum ) / Z.getElement( j, j ) ↩
              );
35      }
36    }
37  }
```

Listing A.2: Implementation of the Crout method on the CPU using matrix $\mathbb{Z}$. Taken from the Decomposition project repository on GitLab[4].

Appendix B

Random Dense Matrix Generator Script

The Python script used to generate random dense matrices that were later decomposed in the benchmark can be seen in Listing B.1.

```python
import numpy as np

num_mtx_files = 3
num_rows_cols_range = [2000, 5000]
elements_range = [-1000, 1000]

mtx_files = np.random.randint(low=num_rows_cols_range[0], high=num_rows_cols_range[1], size=num_mtx_files)

for num_rows_cols in mtx_files:
  dense_mtx_filename = f"Cejka{num_rows_cols}.mtx"

  header_lines = [
  "%%MatrixMarket matrix coordinate real general",
  "%-------------------------------------------------------------------------------",
  "% Cejka Dense Matrix Collection, Lukas Cejka",
  "% Insert URL here",
  f"% name: Cejka/{dense_mtx_filename}",
  "% date: 2022",
  "% author: L. Cejka",
  "% kind: Random Dense Matrix",
  "%-------------------------------------------------------------------------------",
  ]
  header_lines = [line + "\n" for line in header_lines]

  mtx_info = f"{num_rows_cols} {num_rows_cols} {num_rows_cols*num_rows_cols}\n"

  rand_floats = np.random.uniform(low=elements_range[0], high=elements_range[1], size=(num_rows_cols*num_rows_cols))
  rand_floats = [flt+1 if flt == 0 else flt for flt in rand_floats]
  rand_floats = [str(flt) + "\n" for flt in rand_floats]


  with open(dense_mtx_filename, 'w') as f:
    f.writelines(header_lines)
    f.write(mtx_info)
    for col in range(1, num_rows_cols):
```

```
36        for row in range(1, num_rows_cols):
37          f.write(f"{row} {col} {rand_floats[(col-1)*num_rows_cols + row - 1]}")
```

Listing B.1: Python script for generating random dense matrices in the *SuitSparse Matrix Collection* [41] format. On line 28 it can be seen that any zeros were incremented which was done to assure that the matrices produced would be both strongly regular and composed only of nonzero elements.