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 2016/2017

naskenované zadání práce (originál s podpisy a razítkem!)

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**Jakub Fiktivní**
Studijní program:	**Aplikace přírodních věd**
Obor:	**Aplikace softwarového inženýrství**
Název práce česky:	**Webová aplikace pro správu veřejné knihovny s pobočkami**
Název práce anglicky:	**Web Application for Management of Public Library with Branch Offices**

Pokyny pro vypracování:

1. Seznámit se s chodem veřejné knihovny z pohledu zaměstnance a čtenáře.
2. Prozkoumat existující řešení (nabízené funkce, výhody, nevýhody).
3. Na základě výše získaných informací identifikovat typy uživatelů a jim odpovídající funkce webové aplikace.
4. Navrhnout databázi (ERA model) a vybrat vhodný databázový systém, ve kterém bude realizována.
5. Navrhnout vzhled webových stránek tak, aby obsahovaly vhodné funkce pro jednotlivé typy uživatelů, vybrat vhodný framework pro realizaci webové aplikace a implementovat ji s ohledem na známé bezpečnostní hrozby.
6. Vytvořit uživatelskou příručku pro jednotlivé typy uživatelů (včetně popisu instalace a požadovaného softwaru).

Doporučená literatura:

[1] *Doctrine Project* [online]. Doctrine Team, 2006 [cit. 2016-10-19]. Dostupné z: `http://www.doctrine-project.org/`.

[2] *PHP: Hypertext Preprocessor* [online]. Dánsko: The PHP Group, 2001 [cit. 2016-10-19]. Dostupné z: `http://php.net/`.

[3] *Rychlý a pohodlný vývoj webových aplikací v PHP – Nette Framework* [online]. Praha: Nette framework, 2008 [cit. 2016-10-19]. Dostupné z: `https://nette.org/`.

Jméno a pracoviště vedoucího práce:

Mgr. Dana Majerová, Ph.D.
České vysoké učení technické v Praze
FJFI, detašované pracoviště Děčín
Pohraniční 1288/1
405 01 Děčín

Jméno a pracoviště konzultanta:

—

s podpisem vedoucího práce

vedoucí práce

Datum zadání bakalářské práce: 20. 10. 2016

Termín odevzdání bakalářské práce: 10. 7. 2016

Doba platnosti zadání je dva roky od data zadání.

s podpisem vedoucího katedry

vedoucí katedry

s podpisem děkana fakulty

děkan

V Praze dne 20. 10. 2016

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. Tomáš Oberhuber, Ph.D. za vedení mé práce a za podnětné návrhy, které ji obohatily.

Acknowledgment

I would like to thank doc. Ing. Tomas Oberhuber, Ph.D. for supervising my project and for the inspiring proposals that enriched it.

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: Popis práce česky

Klíčová slova: Klíčová slova

Title:
Parallel LU Decomposition for the GPU

Author: Bc. Lukáš Čejka

Abstract: Description of the project in English

Key words: Key words

Contents

Introduction TODO

Put my introduction text here (1-3 pages, do not divide it into sub-pages).

Chapter 1

Theory TO

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 TO

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) TO

As described in section 1.2 of the author's bachelor thesis *Formats for storage of sparse matrices on GPU* [35], 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 [33].

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 [34].

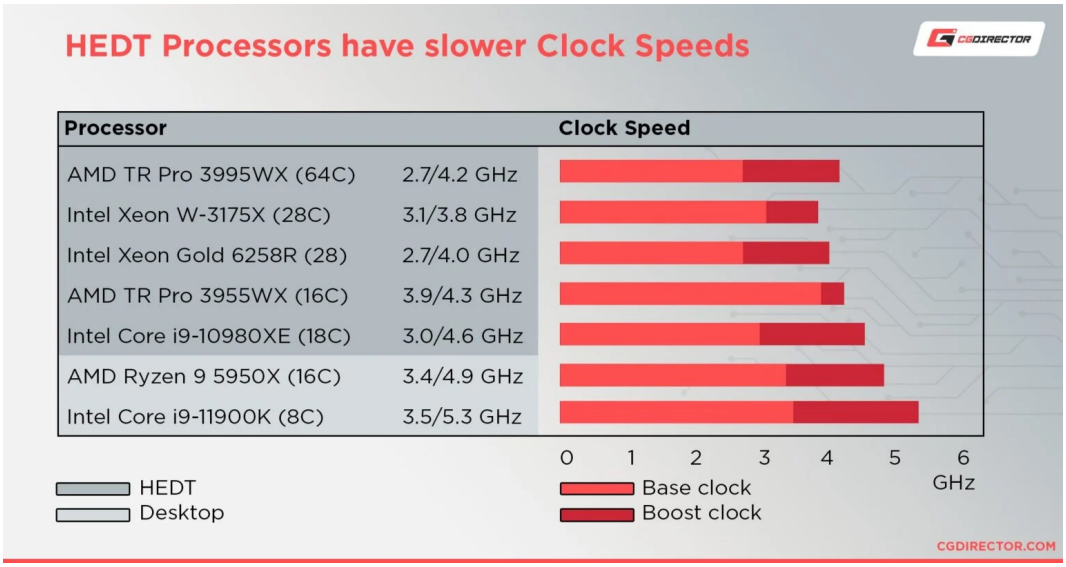


Figure 1.1: Selection of desktop and server CPUs - including number of cores and clock speed [34]. HEDT stands for High-End Desktop Processor - in this instance it also includes CPUs predominantly used in servers.

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 number of threads running at once). Simply, the CPU is more suitable for rapid completion of serialized instructions.

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 for it to compensate for memory access delays by performing computations simultaneously [33].

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 subsection 1.2.6.

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

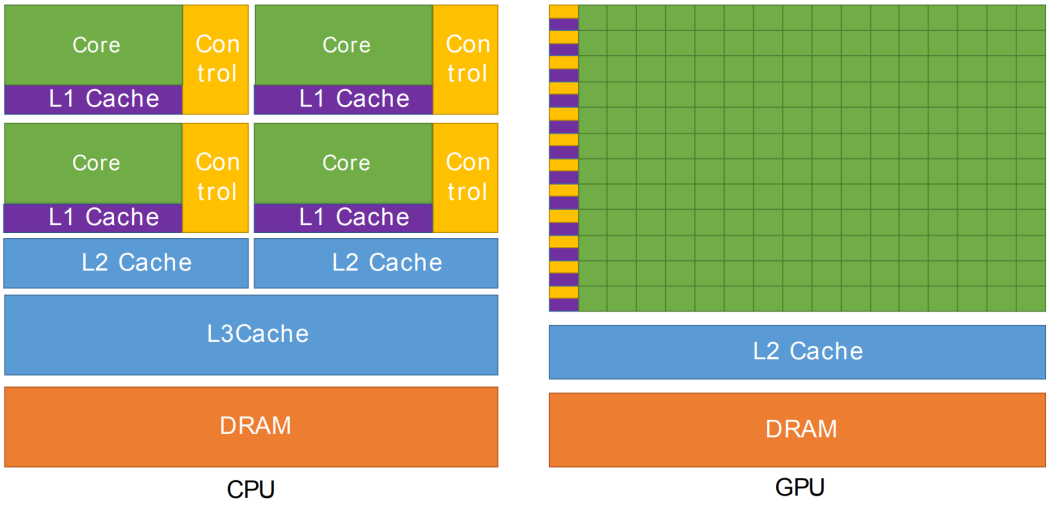


Figure 1.2: Comparison of the architecture of CPUs and GPUs. On one hand, CPUs have fewer cores, more complex controlling logic compared to GPUs and the cores of a CPU have a higher clock speed. On the other hand, GPUs have a much higher number of cores clocked at lower speeds. Taken from Nvidia's *CUDA C++ programming guide* [33].

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, TDP. The value that clearly 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) TO

The Compute Unified Device Architecture (CUDA) is programming model, sometimes referred to as a parallel computing platform, introduced by Nvidia in 2006

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	5120	6912
FP64 Cores / SM	NA	NA	32	
FP64 Cores / GPU (excl. Tensor)	NA	NA	2560	3456
CUDA Cores / SM	128	128	NA	NA
CUDA Cores / GPU	1920	3584	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[3]
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 [29, 30, 25, 26] the RTX 3060 [27, 28, 24, 23] and the A100 [31, 22].

[32]. It is designed to give developers low-level access to GPU hardware, for example, fine-tuning 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 [21].

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 in a similar manner: 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 TO

CUDA introduces many unique concepts that come with their own naming scheme. This subsection will list and explain some basic terminology that will be used throughout the project [16]:

- **Host** - Central Processing Unit (CPU) and its memory. The host provides data to 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** - 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 is executed in parallel across a number of threads.

1.2.2 Thread Management TO

This subsection 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 [33]. 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* [35].

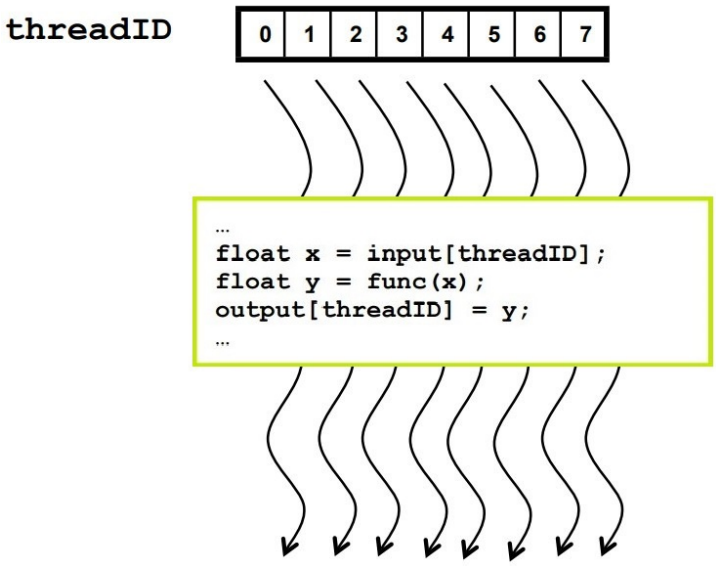


Figure 1.3: On an Nvidia GPU, eight threads with IDs from `0` to `7` execute the code in this example. A value from the array `input` with `threadID` as the index (i.e. each thread will read a value at a different index) is changed using the function `func(x)` and then stored in the same order as before into the array `output`. Taken from *Format for storage of sparse matrices on GPU* [35, 16].

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 comprises of multiple threads executing the same computations on different data items [20]. 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 1 thread, then CUDA will still allocate 32 threads with 31 being inactive. Similarly, if a subset of threads from 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 purpose of the explanation and as such contains only 8 threads.

Block It is not always necessary to use such granularity that warps are able to provide. Developers can choose to execute code on a group of up to 1024 threads

[0]L. Durant, O. Giroux, M. Harris: *Inside Volta: The World's Most Advanced Data Center GPU*. URL: `https://devblogs.nvidia.com/inside-volta/`

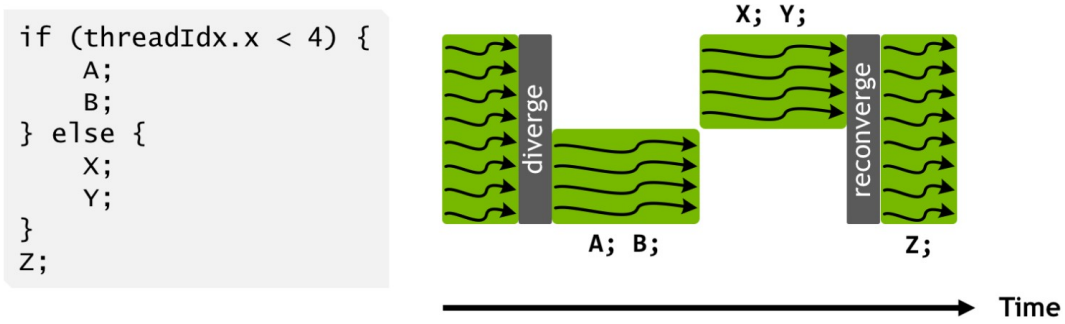


Figure 1.4: Conditional code execution by a warp of 8 threads; each thread is represented by one wavy line. Due to the condition, threads with an ID between `0` and `3` will only execute statements `A` and `B`. However, simultaneously, threads with IDs between `4` and `7` will be put on hold. Comparably, 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].

(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` · `num_threads_y_dim` · `num_threads_z_dim` ≤ 1024) [18, 33].

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 subsection 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 of 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.

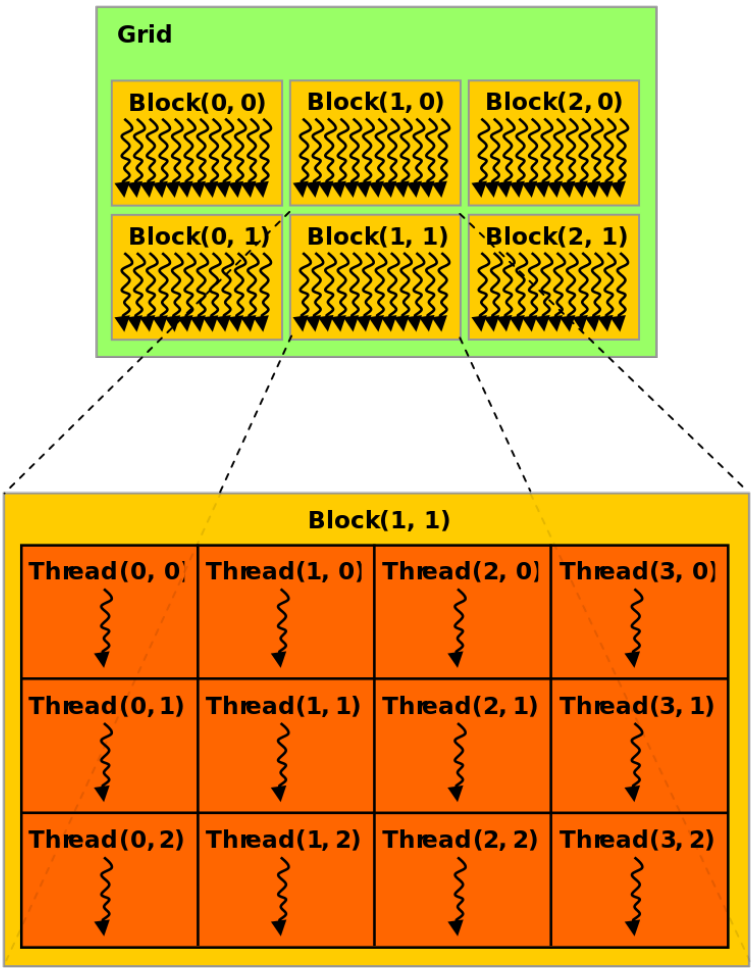


Figure 1.5: CUDA thread structuring of a 2D grid made up of 2D blocks of threads. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

1.2.3 Memory Management TO

The aim of this subsection 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 a number of threads every thread has its own unique ID stored in a variable that is available within the kernel. This variable is stored in memory that 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, with the exception of [33]:

- 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 [33]. 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* [33].

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.

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

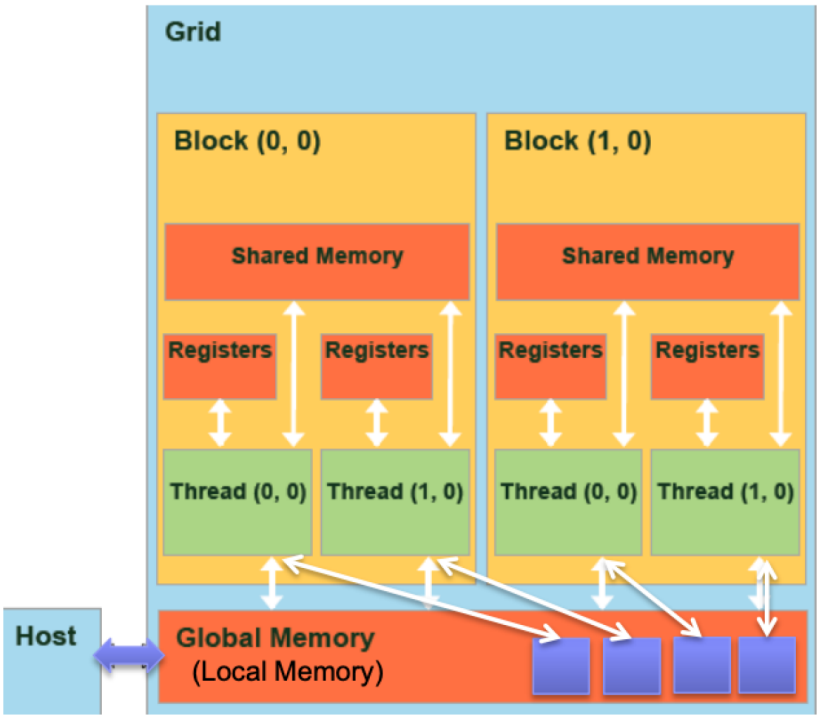
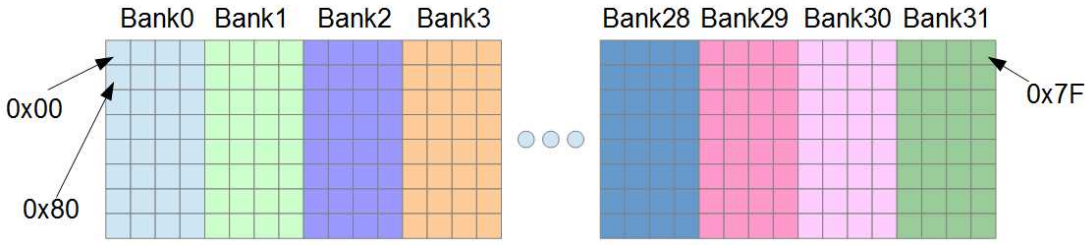


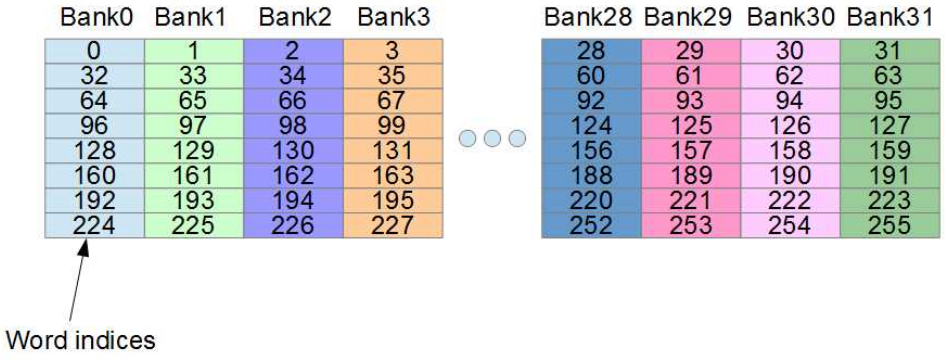
Figure 1.6: Different CUDA memory types visualized. Each thread has access to its local memory comprised of registers, local memory (blue squares in global memory), shared memory of its block, and global memory. Taken from Yao Hsiao's *GPU - CUDA introduction* [15].

can be stored in 32 bits. Apart from words, there are 32 banks per block. Each bank contains a number of words depending on the size of 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.
As mentioned by Chris Rose in *CUDA Succinctly* [14], this division of words into banks is salient as each bank can serve only 1 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



(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 show in Figure). The four bytes to the right of it (first row under `Bank1`) make up the `1st` word. Memory addresses of some bytes are shown hexadecimal.



(b) Shared memory 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 1 byte is 1 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* [14].

is used when the same word from any particular bank is accessed by more than 1 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 of 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 1 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 firstly serve thread `0` with word `0` and then thread `1` with word `32`.
In order to facilitate further understanding, examples from *Cuda succinctly* [14] and

CUDA C++ Programming Guide [33] 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 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.

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 2 threads read from the same bank
`arr[tID*2]`	Slow - 2-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* [14].

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 [13]. 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 of the program. In order 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, de-allocate (free) the data from the device [13] once it is not required.
As mentioned above, in Paragraph *Per-thread local memory* in Subsection 1.2.3, for the purpose of 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*

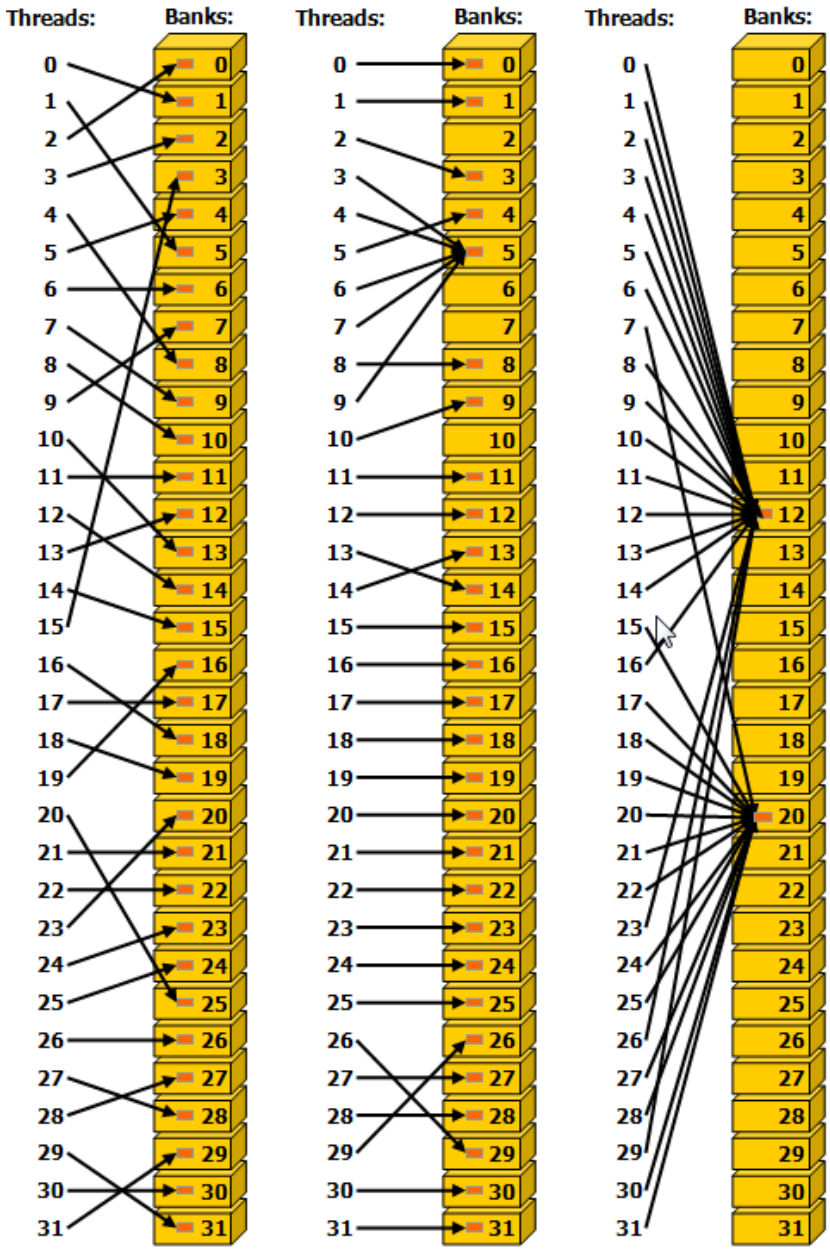


Figure 1.8: Irregular shared memory accesses on 3 separate examples; yellow rectangular cuboids are banks; small orange rectangles are words. The left sub-image shows threads of a warp accessing shared memory randomly, however, in such a way that does not cause bank conflicts. The middle sub-image shows 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 2 broadcast operations. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

Efficiently in CUDA C/C++ Kernels [13]: "*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

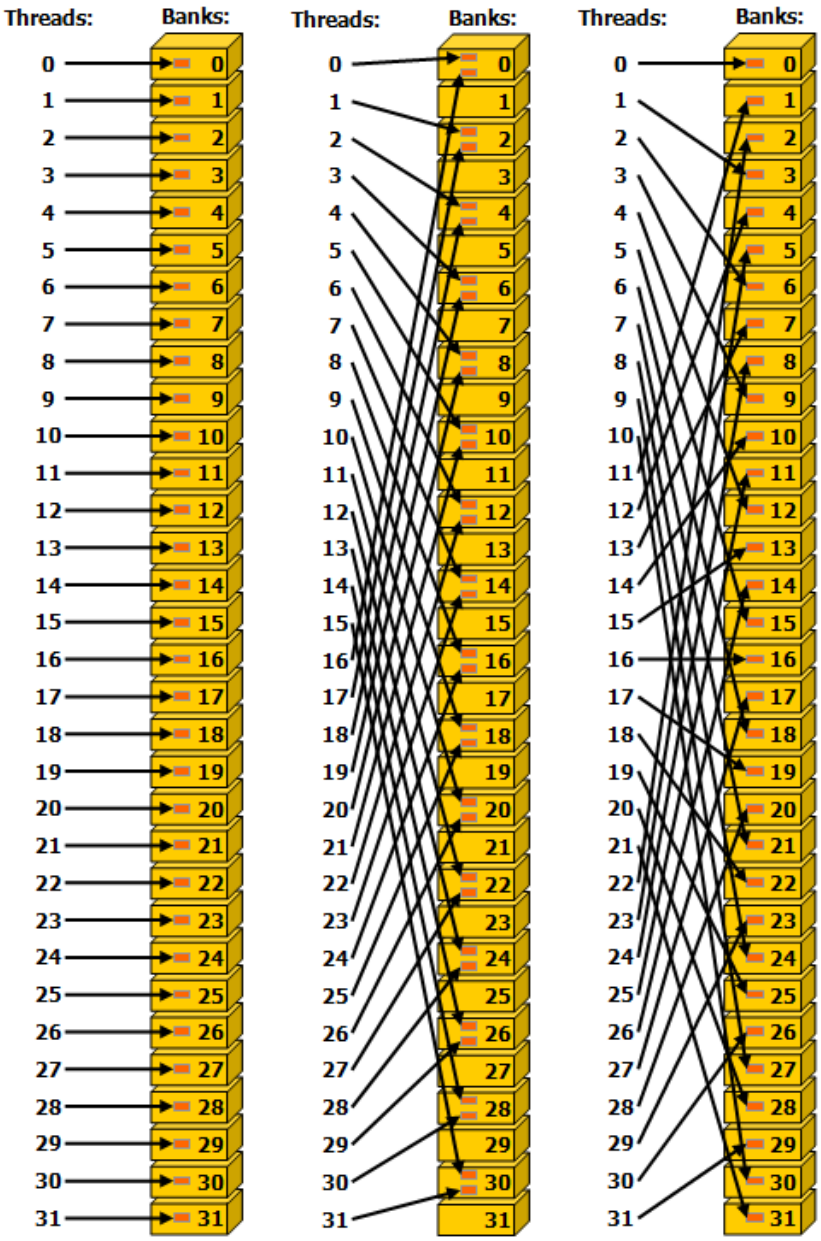


Figure 1.9: Shared memory access using striding on 3 separate examples; yellow rectangular cuboids are banks; small orange rectangles are words. The left sub-image shows threads of a warp accessing shared memory with a stride of 1 (no bank conflict). The middle sub-image shows shared memory accessed by threads with a stride of 2 (equivalent to `arr[tID*2]` from Table 1.3; 2-way bank conflict as threads access 2 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* [33].

types of memory. In order to minimize this bottleneck, it is advised by Nvidia to keep the number of transactions that require accessing global memory to as few as possible [13]. One of the main concepts that achieves this is called *global memory coalesced access*. This methodology takes full advantage of the SIMD approach,

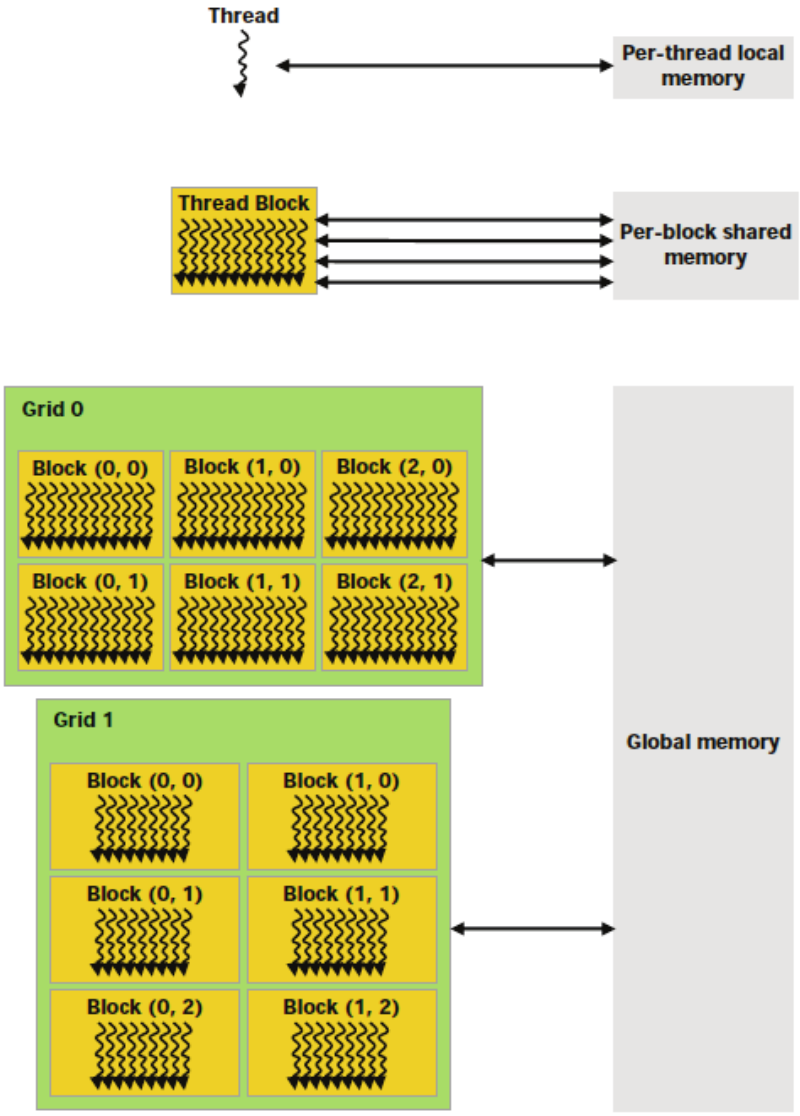


Figure 1.10: CUDA memory structuring. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

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 [12].
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 [10, 33]:

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 - 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 1 memory transaction.
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 [33]:

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

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* [33].

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 2 components: `(x, y)`. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

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;
```

```
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) is not coalesced, therefore, 4 bytes are used for padding. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

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 be fully coalesced, which leads to lower bandwidth and sub-optimal performance.
Examples of coalesced and non-coalesced memory accesses by a 16-thread warp are shown in Figures 1.11 and 1.12.

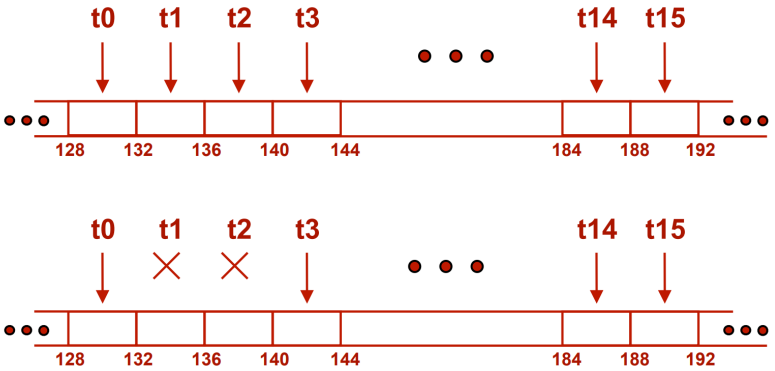
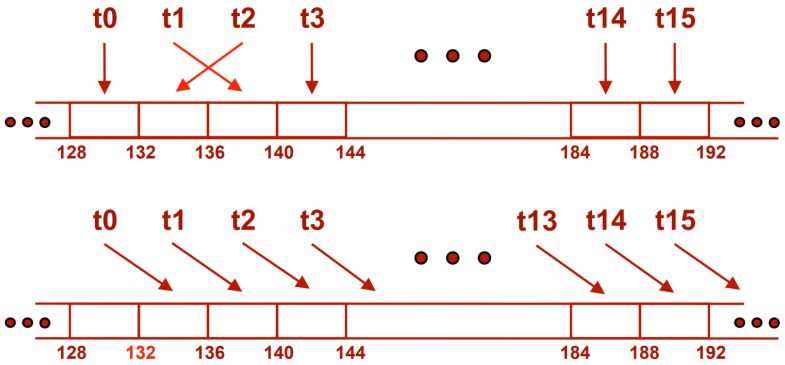


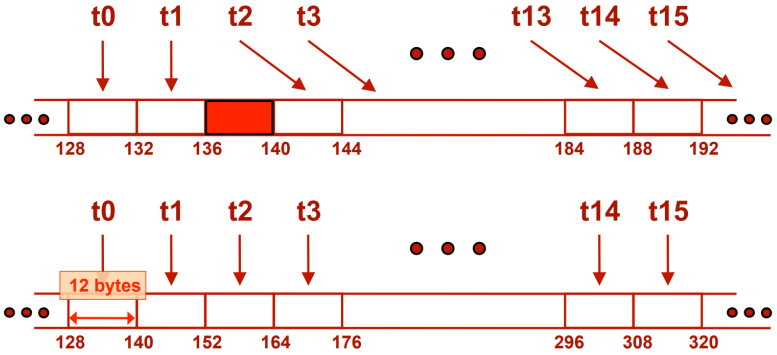
Figure 1.11: Coalesced access to global memory. Threads are denoted with `t<ID>` and the numbers below the rectangular memory locations are global memory addresses. In the upper image, all threads of the warp access successive memory addresses which house words of size 4 bytes (`float`), the size of 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 2 threads from the 16-thread warp do not access data. Thus, threads of the warp have 2 separate execution paths (thread/warp divergence) that are executed sequentially. However, from a memory standpoint, the access is still coalesced as all requirements are fulfilled. Taken from Martínez Manuel Ujaldón's *CUDA Optimizations, Debugging and Profiling* [10].

In summary, coalesced access (reading and writing) to global memory must be fulfilled in order 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



(a) In the upper image, 2 threads swap orders and thus do not access successive floats in global memory. However, according to *CUDA C++ Programming Guide* [33] 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 2 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 Sub-figure 1.12a the memory transaction will be split into 2 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 and 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 1 exception being 12-byte structures. Taken from Martínez Manuel Ujaldón's *CUDA Optimizations, Debugging and Profiling* [10].

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 de-allocated 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* [33] 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 Subsection 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).
- 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 [33].

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()` [33].
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 Subsection 1.2.4) is used [33]. These problems can be avoided by forcing memory 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* [33].

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

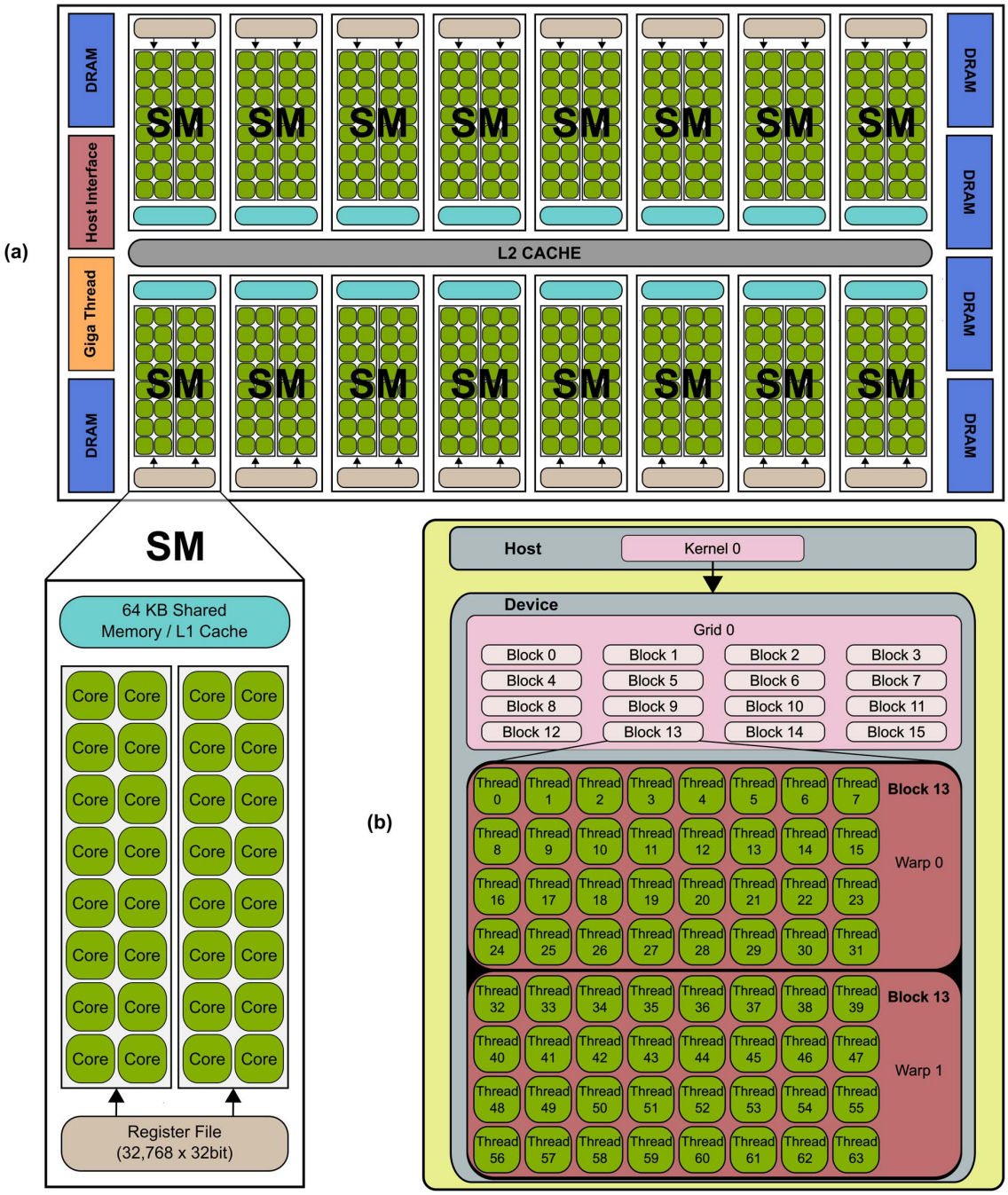


Figure 1.13: a) Usual architecture of an Nvidia Fermi GPU that comprises of SMs. Furthermore, each SM is made up of SP cores (Stream Processor cores) b) CUDA programming model controls the hardware of the GPU. Taken from *Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs* [6].

1.2.4 Asynchronous concurrent execution TO

Within CUDA there are multiple operations that can be executed at the same time (concurrently) - namely [33]:

- Computation on the host
- 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* [33] 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 subsection 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 job 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 [33].
Host asynchrony is available for the following device operations [33]:

- 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 [33].

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* [33].

The second limitation - CUDA context - signifies that if there are two CUDA applications running at the same time on a system, then two kernels - one from each context - will not be run in 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 the previous 2 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 1 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 [33]. In terms of threads, the limiting factor is the maximum number of active threads - described at the end of the *Grid* paragraph in Subsection 1.2.2. For example, if 2 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 TO

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" [17]. To put it another way, a single stream is basically 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 resource available (active threads and memory). Therefore, multiple active streams 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 Section1.2.4.
Unlike individual kernels, global memory is used by all streams without division, i.e. all streams have access to the same global memory and there is no part of global memory that would belong to a particular stream.
According to Nvidia's *CUDA C++ Programming Guide* [33], individual streams are not dependent on each other, meaning that they are able to execute commands separately or concurrently. However, this behavior does not have to be consistent, for example, communication between kernels is not define. 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 synchronicity of streams is required.
This subsection 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.

```
1 // Declare array of 2 stream objects
2 cudaStream_t streams[2];
3
4 // Create each stream
5 for (int i = 0; i < 2; ++i)
6   cudaStreamCreate(&streams[i]);
```

Listing 1.2: Creation of streams. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

In order 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 Subsection 1.2.5. Launching a kernel on a specific stream is shown in Listing 1.3.

```
1 // Launch MyKernelA using the 0th stream using 1 block made up of 1 ↩
     thread and 0 bytes of dynamically allocated shared memory
2 MyKernelA<<<1, 1, 0, stream[0]>>>(inputVariableA)
3
4 // Launch MyKernelB using the 1st stream
```

```
5 MyKernelB<<<1, 1, 0, stream[1]>>>(inputVariableB)
```

Listing 1.3: Pseudo-code for launching 2 different kernels using 2 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 1 single-thread block with 0 bytes of dynamic shared memory allocated, thus, the devic resources will not be exhausted and both kernels will run concurrently. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

When streams are no longer needed, they are to be destroyed using `cudaStreamDestroy()` - show 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.

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

Listing 1.4: Destruction of streams. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

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 instruction are executed in order with respect to the device (not concurrently as shown in the examples above) [33].
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 [33, 9]:

- `cudaDeviceSynchronize()` - Synchronization of the entire device. This function servers 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 a checkpoint to wait the for completion for 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* [9].

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* [33] that if there are 2 streams, then 2 commands - each from 1 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 2 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 [33]:

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

Thus, in order 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 TO

As previously mentioned at the beginning of Subsection 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 the purpose of this project only the core basics that were used during its development along with some other important extensions will be detailed.
This subsection will divided into 2 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 Subsection 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 [33, 9, 35]:

- `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* [9].

- `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 Subsection 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 [33]:

- `__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 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 own 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(void)
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++ ) a_h[ i ] = 100.f + i;
```

```
15
16      // copying data from host -> device -> device -> host
17      cudaMemcpy( a_d, a_h, nBytes, cudaMemcpyHostToDevice );
18      cudaMemcpy( b_d, a_d, nBytes, cudaMemcpyDeviceToDevice );
19      cudaMemcpy( b_h, b_d, nBytes, cudaMemcpyDeviceToHost );
20
21      // checking that all data is equal
22      for( i = 0; i < N; i++ ) assert( a_h[ i ] == b_h[ i ] );
23
24      free( a_h ); free( b_h ); // freeing data on host
25      cudaFree( a_d ); cudaFree( b_d ); // freeing data on device
26      return 0;
27  }
```

Listing 1.5: Example of code that utilizes CUDA memory managing extensions of C++. Taken from *Formats for storage of sparse matrices on GPU* [35] and Nvidia's *Getting Started with CUDA* presentation [16].

Kernels While the term *kernel* was introduced earlier in Subsection 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 [33, 35]:

- `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 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* [33].

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* [33]:

- `__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 functions 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* [33].

Inter-kernel extensions TO

This category comprises of variables, structures and operations etc. that are most often used within a kernel. First, the means by which developers are able to 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 a 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 [33]:

- `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 2 matrices. In such an example, each thread would take 2 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()
{
  ...
```

```
16    // Number of threads per each 2-dimensional block is 16x16 = 256
17    dim3 threadsPerBlock(16, 16);
18
19    // Number of blocks in the grid depends on the matrix dimension (↩
          NxN)
20    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
21
22    // Kernel launch: A + B = C
23    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
24    ...
25 }
```

Listing 1.7: Example of C++ pseudocode of a kernel that adds 2 matrices using 2-dimensional thread blocks. Since each thread has a unique `globalID` for each of 2 dimensions, then those IDs can be used as indices for adding matrix elements. This simple example does not take into account allocation and copying data from host to device. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

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

Finally, as mentioned in paragraph *Global memory* in Subsection 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 in above in paragraph *Memory managing extensions* in Subsection 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 [33]. An example showing this use of the specifier can be seen in Listing 1.11.
In paragraph *Shared memory* in Subsection 1.2.3, it was mentioned that threads of a block are able to 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* [7]. 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 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 their work until this point. To put it more clearly: threads of a block are halted on the the line in code that contains `__syncthreads()` until all threads of the block arrive to at it [33].

1.2.6 Matrix multiplication TO

In order to consolidate how CUDA can be used to accelerate the execution of a simple task, such as matrix multiplication, this subsection will present solutions to

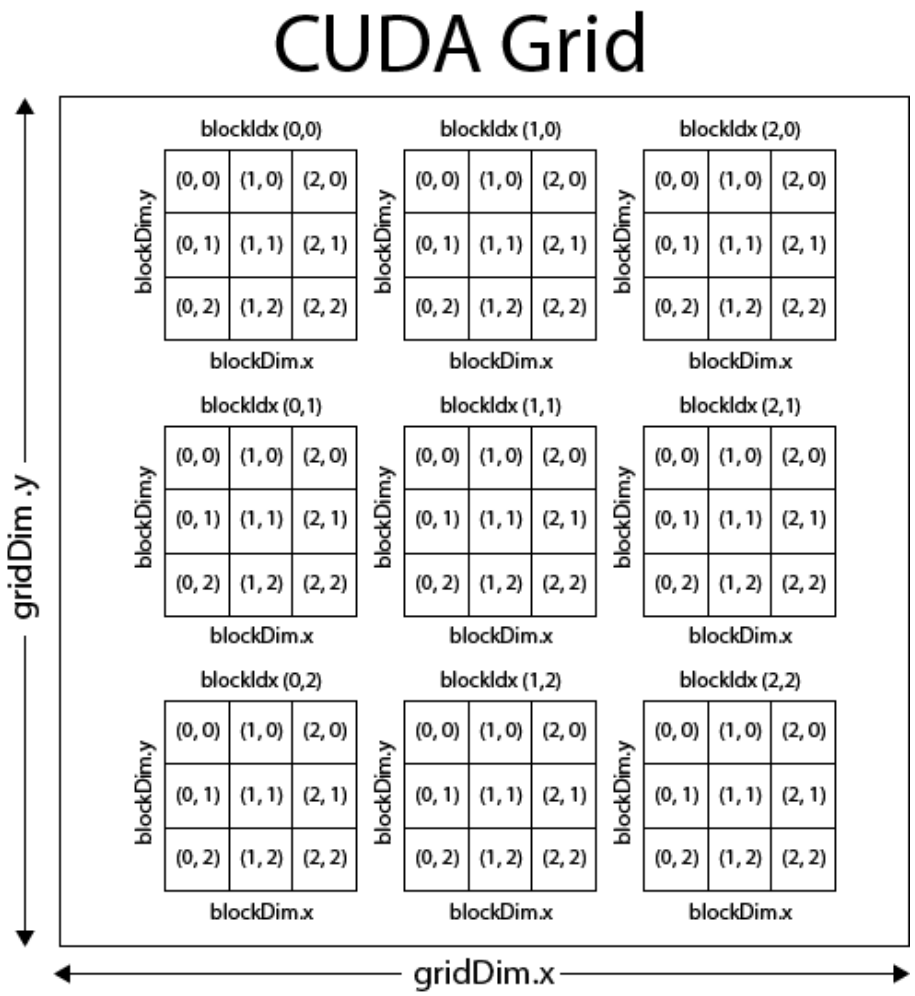


Figure 1.14: Grid made up of 2-dimensional thread blocks. Each square represents a thread with its IDs within its block (`threadIdx.x` , `threadIdx.y`). It can be imagined that each block is a sub-matrix and together all sub-matrices make up the full matrix. Taken from *CUDA Parallel Thread Management* [8].

this task from Nvidia's *CUDA C++ Programming Guide* [33]. Furthermore, for the purpose of showcasing and stressing the importance of abiding by Nvidia's recommendations when it comes to best practices and optimal performance, 2 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* [33]. 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 2 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* [33].

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

1. Take matrices $\mathbb{A}$, $\mathbb{B}$, $\mathbb{C}$ 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 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;
```

```
10    d_A.width = A.width; d_A.height = A.height;
11    size_t size = A.width * A.height * sizeof(float);
12    cudaMalloc( &d_A.values, size );
13    cudaMemcpy( d_A.values, A.values, size, cudaMemcpyHostToDevice );
14
15    // Allocate and copy B to device memory
16    Matrix d_B;
17    d_B.width = B.width; d_B.height = B.height;
18    size = B.width * B.height * sizeof(float);
19    cudaMalloc( &d_B.values, size );
20    cudaMemcpy( d_B.values, B.values, size, cudaMemcpyHostToDevice );
21
22    // Allocate C in device memory
23    Matrix d_C;
24    d_C.width = C.width; d_C.height = C.height;
25    size = C.width * C.height * sizeof(float);
26    cudaMalloc( &d_C.values, size );
27
28    // Invoke kernel
29    dim3 threadPerBlock( BLOCK_SIZE, BLOCK_SIZE );
30    dim3 numBlocks( B.width / threadPerBlock.x, A.height / ↩
          threadPerBlock.y );
31    MatMulKernel<<< numBlocks, threadPerBlock >>>( d_A, d_B, d_C );
32
33    // 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* [33].

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

Example without shared memory TO

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 thread such that each thread is responsible for calculating exactly 1 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.

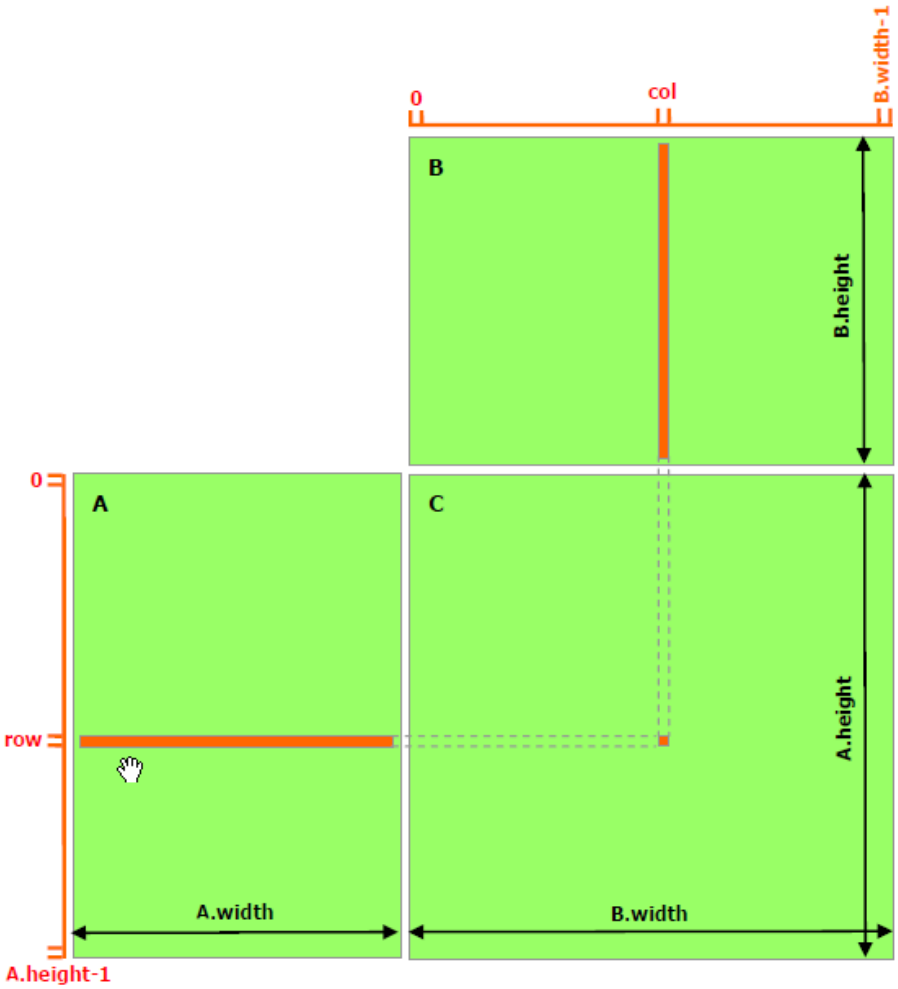


Figure 1.15: Visualization of the execution of the kernel that does not use shared memory - shown in Listing 1.10. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel( Matrix A, Matrix B, Matrix C )
{
  // Each thread computes one element of C by accumulating results ↩
     into Cvalue
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for( int i = 0; i < A.width; ++i )
    Cvalue += A.values[row * A.width + i] * B.values[i * B.width + ↩
       col];

  C.values[row * C.width + col] = Cvalue;
}
```

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* [33].

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 be coalesced for the threads within each warp. Nevertheless, this kernel is sub-optimal as accessing global memory is considered sub-optimal when avoidable.

At the end of the introduction to Subsection 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 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 TO

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 individual element to a block of threads iterating over sub-matrices of `A` and `B`.

In order to make use of shared memory, matrix `C` is divided into sub-matrices of `BLOCK_SIZE x BLOCK_SIZE` elements ($\mathbb{C}_{sub}$). Then, each sub-matrix $\mathbb{C}_{sub}$ 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 an `1xn` and `nx1` matrix. The approach with shared memory expands this concept to blocks. In other words, the sub-matrix $\mathbb{C}_{sub}$ is a result of multiplying 2 rectangular sub-matrices of `A` and `B` (illustrated in Figure 1.16) - denoted $\mathbb{A}_{rect}$ and $\mathbb{B}_{rect}$ for the purpose of this explanation:

1. $\mathbb{A}_{rect}$ is a `BLOCK_SIZE x A.width` sub-matrix of `A` that has the same row indices as $\mathbb{C}_{sub}$.

2. $\mathbb{B}_{rect}$ is a `B.height x BLOCK_SIZE` sub-matrix of `B` that has the same column indices as $\mathbb{C}_{sub}$.

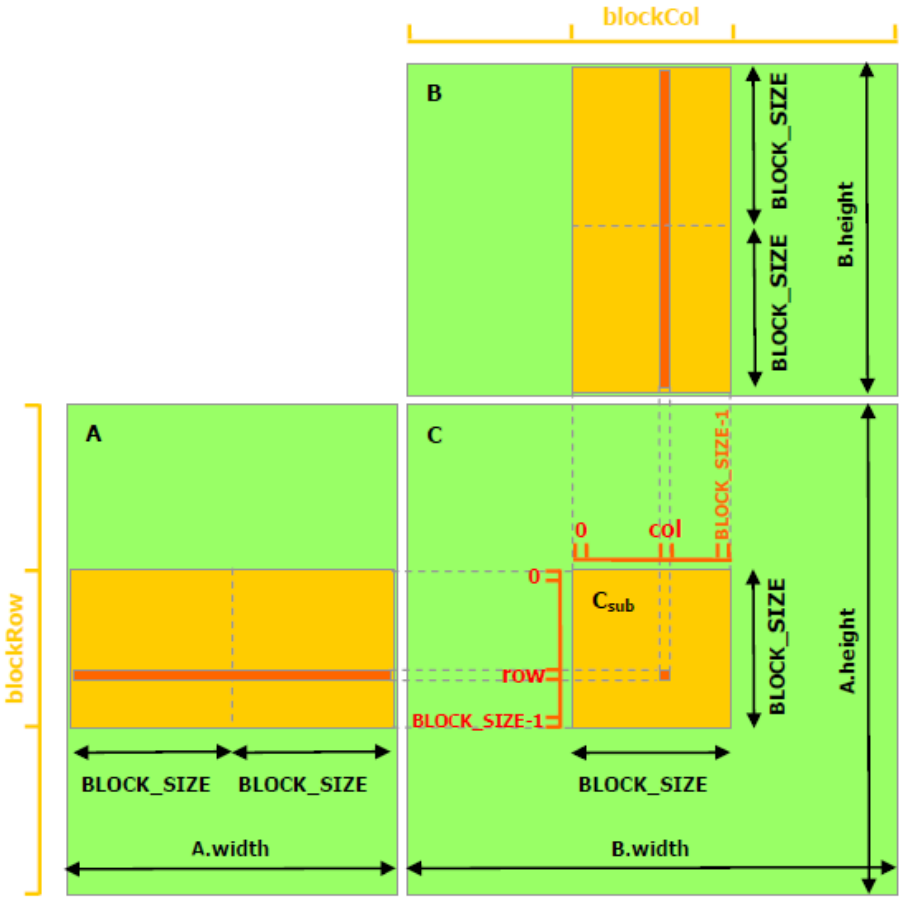


Figure 1.16: Visualization of the execution of the kernel that uses shared memory - shown in Listing 1.11. Taken from Nvidia's *CUDA C++ Programming Guide* [33].

The rectangular sub-matrices $\mathbb{A}_{rect}$ and $\mathbb{B}_{rect}$ are split into `BLOCK_SIZE x BLOCK_SIZE` sub-matrices $\mathbb{A}_{sub}$ 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 $\mathbb{C}_{sub}$ sub-matrix. The computation of $\mathbb{C}_{sub}$ is finished once the thread block reaches and multiplies the rightmost $\mathbb{A}_{sub}$ and the lowermost $\mathbb{B}_{sub}$ sub-matrix.
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 a $\mathbb{A}_{sub}$ 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're responsible 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.

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel( Matrix A, Matrix B, Matrix C )
{
  // Block row and column - indices of the sub-matrices within A and ↩
     B respectively
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;

  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix( C, blockRow, blockCol );

  // Each thread computes one element of Csub by accumulating results↩
      into Cvalue
  float Cvalue = 0;

  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;

  // Loop over all the sub-matrices of A and B that are required to ↩
     compute Csub by multiplying each pair of sub-matrices together ↩
     and accumulate the results
  for( int i = 0; i < (A.width / BLOCK_SIZE); ++i ) {

    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix( A, blockRow, i );

    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix( B, i, blockCol );

    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement( Asub, row, col );
    Bs[row][col] = GetElement( Bsub, row, col );

    // Synchronize to make sure the sub-matrices are loaded before ↩
       starting the computation
    __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      // Synchronize to make sure that the preceding computation is ↩
            done before loading two new sub-matrices of A and B in the ↩
            next iteration
43      __syncthreads();
44    }
45
46    // Write Csub to device memory - each thread writes one element
47    SetElement( Csub, row, col, Cvalue );
48  }
```

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 returns a `BLOCK_SIZE x BLOCK_SIZE` sub-matrix of a matrix that is located `col` sub-matrices to the right and `row` sub-matrices down from the upper-left corner of a 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* [33].

Unlike the previous example, the code shown in Listing 1.11 has less accesses to global memory which leads to an performance improvement as detailed later in the *Comparison of examples* part of Subsection 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 sub-matrices $\mathbb{A}_{sub}$ and $\mathbb{B}_{sub}$ into shared memory. This means that all threads will perform `BLOCK_SIZE` times less accesses to global memory and `A.width + B.height` accesses to shared memory instead. Equivalently to the example before, the access to global memory is coalesced. Thus, by Nvidia's recommendations, this kernel can be considered being 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 si done, terminate the threads that would reach out of 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 TO

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 16000 by 16000. The hardware and software specifications of the machine used for the benchmarks can be found in Table 1.6. Furthermore, the full code can be found in Attachment A.

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 was 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 16000 by 16000.

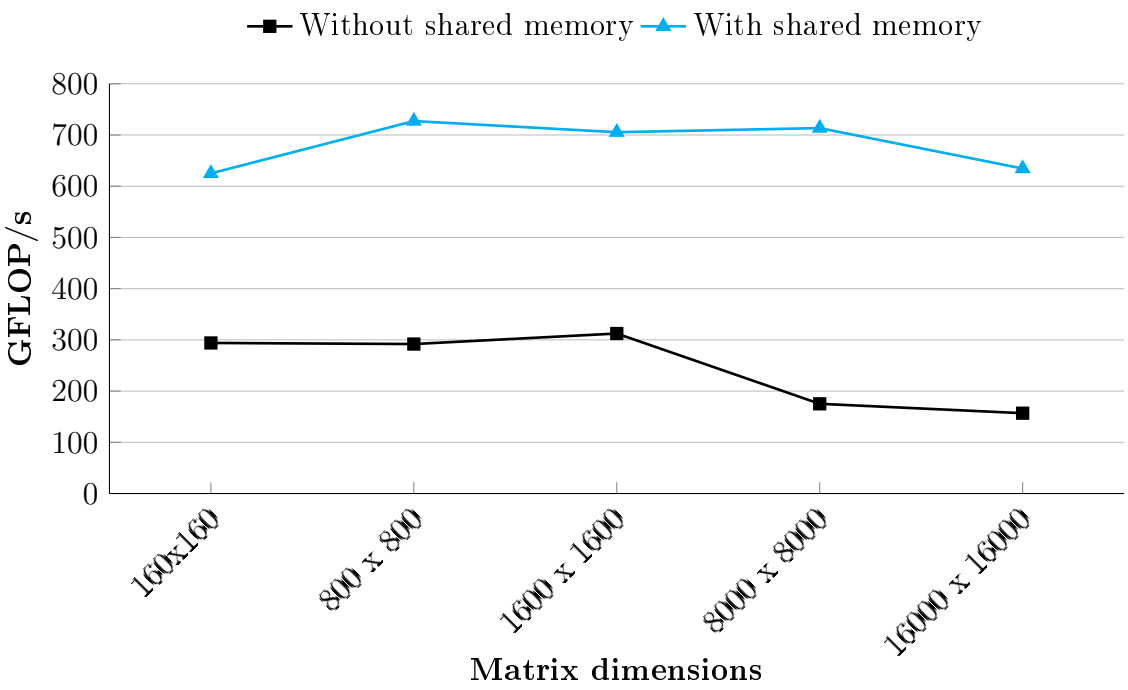


Figure 1.17: Matrix multiplication results of matrices with varying dimensions; matrices are filled with 1 element. The values - presented in GFlop/s - are averages from 10 loops of the operation.

	Results [msec]	
Matrix dimensions	Without shared memory	With shared memory
160 x 160	0.028	0.013
800 x 800	3.509	1.408
1600 x 1600	26.224	11.613
8000 x 8000	5846.247	1435.221
16000 x 16000	52226.867	12910.967

Table 1.7: Matrix multiplication results of matrices with varying dimensions; matrices are filled with 1 element. The values - presented in milliseconds - are averages from 10 loops of the operation.

1.3 LU Decomposition TO

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, 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 re-

quirement 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 of 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 re-written 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 a 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 1 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 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* [1].

In summary, from the steps above, it can 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 TO

This subsection 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* [5].
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 filling in 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 filling-in 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 pseudo-code implementing the algorithm can be seen in Listing 1.12 and the visualization of the algorithm's advance is shown in Figure 1.18.

```
void crout_method(A, L, U) {
  int i, j, k;
  double sum = 0;

  // Fill main diagonal of U with 1s
  for (i = 0; i < n; i++) {
    U[i][i] = 1;
  }

  // Loop through the main diagonal
  for (j = 0; j < n; j++) {

    // Compute column j in L
    for (i = j; i < n; i++) {
      sum = 0;
      for (k = 0; k < j; k++) {
        sum += L[i][k] * U[k][j];
      }
      L[i][j] = A[i][j] - sum;
    }

    // Compute row j in U
    for (i = j; i < n; i++) {
      sum = 0;
      for(k = 0; k < j; k++) {
        sum = sum + L[j][k] * U[k][i];
      }
      if (L[j][j] == 0) {
        printf("det(L) close to 0!\n Can't divide by 0...\n");
        exit(EXIT_FAILURE);
      }
      U[j][i] = (A[j][i] - sum) / L[j][j];
    }
  }
}
```

Listing 1.12: C++ Pseudo-code 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 reprenting 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* [4], *Numerical recipes: the art of scientific computing* [5] and *Crout matrix decomposition* [3].

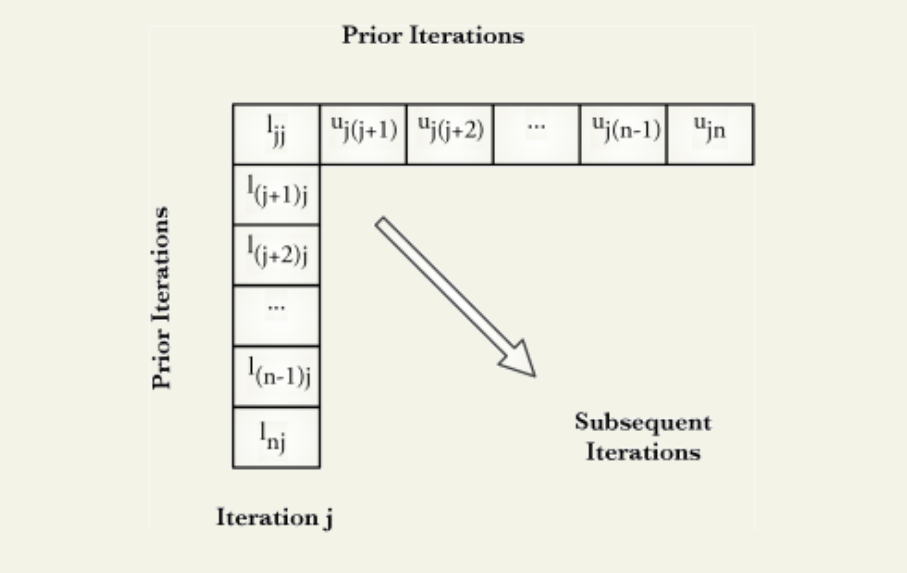


Figure 1.18: Sequence of computations of the Crout method. First, column j in $\mathbb{L}$ is computed and then row j of $\mathbb{U}$. Taken from Vismor's *Crout's LU Factorization* [4].

1.3.2 Numerical method TO

When solving systems of linear equations there are 2 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 Subsection 1.3.1 is a direct method, 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* [36] 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 method belonging to this group as algorithms that do not take every non-zero into account.
However, for the purpose of 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 $\mathbb{A} = \mathbb{L}\mathbb{U}$ 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 $\mathbb{U}^{t+1}$

$$\begin{aligned} l_{ij}^{t+1} &= a_{ij} - \sum_{k=1}^{j-1} l_{ik}^t u_{kj}^t && i \geq j\,, \\ u_{ij}^{t+1} &= \frac{1}{l_{ii}^t}\left(a_{ij} - \sum_{k=1}^{i-1} l_{ik}^t u_{kj}^t\right) && i < j \end{aligned} \tag{1.14}$$

.

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

4. If both differences are smaller than the 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 convergence 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 TODO

2.1 Project TODO

2.1.1 LU Decomposition TODO

2.1.2 Unit Tests TODO

2.1.3 Benchmarks TODO

2.2 Optimization TODO

2.3 LU Decomposition TODO

Chapter 3

Benchmark results TODO

Conclusion TODO

Put my conclusion text here (1-3 pages, do not divide it into sub-pages) or insert it from a separate file using: `\input{conclusion.tex}`.

Bibliography

[1] LINDFIELD, George a John PENNY. Linear Equations and Eigensystems. *Numerical Methods* [online]. Elsevier, 2019, 2019, s. 73-156 [cit. 2022-06-28]. ISBN 9780128122563. Dostupné z: doi:10.1016/B978-0-12-812256-3.00011-7

[2] Crout matrix decomposition. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2022-06-28]. Dostupné z: https://en.wikipedia.org/wiki/Crout_matrix_decomposition

[3] LU decomposition. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2022-06-28]. Dostupné z: https://en.wikipedia.org/wiki/LU_decomposition

[4] VISMOR. 4.3 Crout-s LU Factorization. In: *Vismor* [online]. [cit. 2022-06-28]. Dostupné z: https://vismor.com/documents/network_analysis/matrix_algorithms/S4.SS3.php

[5] PRESS, William H. *Numerical recipes: the art of scientific computing*. 3rd ed. Cambridge: Cambridge University Press, 2007, 50-52. ISBN 9780521880688.

[6] HERNÁNDEZ, Moisés, Ginés D. GUERRERO, José M. CECILIA, et al. Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs. *PLoS ONE* [online]. 2013, **8**(4), 2 [cit. 2022-05-20]. ISSN 1932-6203. Dostupné z: doi:10.1371/journal.pone.0061892

[7] HARRIS, Mark. Using Shared Memory in CUDA C/C++. *Nvidia Developer: Technical Blog* [online]. 28 January 2013 [cit. 2022-06-25]. Dostupné z: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[8] MCKENNON, Justin. CUDA Parallel Thread Management. In: *Microway* [online]. 13 June 2013 [cit. 2022-06-23]. Dostupné z: https://www.microway.com/hpc-tech-tips/cuda-parallel-thread-management/

[9] NVIDIA. CUDA Runtime API: API Reference Manual. *Nvidia Docs* [online]. January 2022 [cit. 2022-06-21]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf

[10] MARTÍNEZ, Manuel Ujaldón. CUDA Optimizations, Debugging and Profiling. *Partnership for advanced computing in Europe* [online]. [cit. 2022-06-14]. Dostupné z: http://materials.prace-ri.eu/35/1/gpuvideo4.pdf

[11] BROWN, Gordon. ComputeCpp v1.1.6: Changes to Work-item Mapping Optimization. In: *Codeplay* [online]. 18 November 2019 [cit. 2022-06-14]. Dostupné z: https://codeplay.com/portal/blogs/2019/11/18/computecpp-v1-1-6-changes-to-work-item-mapping-optimization.html

[12] CABRERA, Fang. The CUDA Parallel Programming Model - 5. Memory Coalescing. In: *Fan Cabrera: A tech notebook* [online]. 4 December 2019 [cit. 2022-06-14]. Dostupné z: https://nichijou.co/cuda5-coalesce/

[13] HARRIS, Mark. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. In: *Nvidia Developer: Technical Blog* [online]. 7 January 2013 [cit. 2022-06-14]. Dostupné z: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[14] ROSE, Chris. *Cuda Succinctly*. United States: CreateSpace Independent Publishing Platform, 2017. ISBN 9781542827409.

[15] HSIAO, Yao. GPU: CUDA intro. *Hack MD* [online]. 17 December 2019 [cit. 2022-06-11]. Dostupné z: https://hackmd.io/@yaohsiaopid/ryHNKkxTr?type=view

[16] RUETSCH, Greg a Brent OSTER. Getting Started with CUDA. In: *Nvidia* [online]. 2008 [cit. 2022-06-07]. Dostupné z: https://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Train

[17] RENNICH, Steve a Nvidia. CUDA C/C++ Streams and Concurrency. *Nvidia Developer* [online]. [cit. 2022-06-07]. Dostupné z: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar

[18] ABI-CHAHLA, Fedy. Nvidia's CUDA: The End of the CPU?. *Tom's Hardware* [online]. 18 June 2008 [cit. 2022-06-05]. Dostupné z: https://www.tomshardware.com/reviews/nvidia-cuda-gpu,1954.html

[19] DURANT, Luke, Olivier GIROUX, Mark HARRIS a Nick STAM. Inside Volta: The World-s Most Advanced Data Center GPU. In: *Nvidia Developer: Technical Blog* [online]. 10 May 2017 [cit. 2022-05-30]. Dostupné z: https://developer.nvidia.com/blog/inside-volta/

[20] MARZIALE, Lodovico, Santhi MOVVA, Golden G. RICHARD III, Vassil ROUSSEV a Loren SCHWIEBERT. Massively Threaded Digital Forensics Tools. LI, Chang-Tsun, ed. *Handbook of Research on Computational Forensics, Digital Crime, and Investigation* [online]. IGI Global, 2010, 2010, s. 234-256 [cit. 2022-05-30]. Advances in Digital Crime, Forensics, and Cyber Terrorism. ISBN 9781605668369. Dostupné z: doi:10.4018/978-1-60566-836-9.ch010

[21] , dogma1138. CUDA support much more languages than just C++ and Fortran. In: *Hacker News* [online]. 27 March 2021 [cit. 2022-05-22]. Dostupné z: https://news.ycombinator.com/item?id=26605219

[22] , Nvidia. NVIDIA A100 TENSOR CORE GPU: Unprecedented acceleration at every scale. In: *Nvidia* [online]. 2022 [cit. 2022-05-22]. Dostupné z: https://www.nvidia.com/en-us/data-center/a100/

[23] MAY, Keith. NVIDIA GeForce RTX 3060 Ti Founders Edition Graphics Card Review. In: *WCCFtech* [online]. 1 December 2020 [cit. 2022-05-22]. Dostupné z: https://wccftech.com/review/nvidia-geforce-rtx-3060-ti-founders-edition-graphics-card-review/2/

[24] , TechPowerUp. NVIDIA GeForce RTX 3060. In: *TechPowerUp* [online]. [cit. 2022-05-22]. Dostupné z: https://www.techpowerup.com/gpu-specs/geforce-rtx-3060.c3682

[25] SMITH, Ryan. NVIDIA Posts Full GeForce GTX 1070 Specifications: 1920 CUDA Cores Boosting to 1.68GHz. In: *AnandTech* [online]. 18 May 2016 [cit. 2022-05-22]. Dostupné z: https://www.anandtech.com/show/10336/nvidia-posts-full-geforce-gtx-1070-specs

[26] , TechPowerUp. NVIDIA GeForce GTX 1070. In: *TechPowerUp* [online]. [cit. 2022-05-22]. Dostupné z: https://www.techpowerup.com/gpu-specs/geforce-gtx-1070.c2840

[27] WALTON, Jarred. Nvidia GeForce RTX 3060 12GB Review: Hope Springs Eternal. In: *Tom's HARDWARE* [online]. 7 July 2021 [cit. 2022-05-22]. Dostupné z: https://www.tomshardware.com/reviews/nvidia-geforce-rtx-3060-review

[28] , W1zzard. MSI GeForce RTX 3060 Gaming X Trio Review: The GeForce Ampere Architecture. In: *TechPowerUp* [online]. 28 February 2021 [cit. 2022-05-22]. Dostupné z: https://www.techpowerup.com/review/msi-geforce-rtx-3060-gaming-x-trio/2.html

[29] HAGEDOORN, Hilbert. Nvidia GeForce GTX 1070 review - Pascal GPU Architecture. In: *The guru of 3D* [online]. 6 October 2016 [cit. 2022-05-22]. Dostupné z: https://www.guru3d.com/articles-pages/nvidia-geforce-gtx-1070-review,3.html

[30] , Nvidia. NVIDIA TURING GPU ARCHITECTURE: Graphics Reinvented. In: *Nvidia* [online]. 2018 [cit. 2022-05-22]. Dostupné z: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[31] , Nvidia. NVIDIA A100 Tensor Core GPU Architecture: UNPRECEDENTED ACCELERATION AT EVERY SCALE. In: *Nvidia* [online]. 2020 [cit. 2022-05-22]. Dostupné z: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[32] OH, Fred. What Is CUDA?. In: *Nvidia Official Blog* [online]. 10 September 2012 [cit. 2022-05-20]. Dostupné z: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

[33] NVIDIA, Corporation. *CUDA C++ Programming Guide* [online]. May 2022 [cit. 2022-05-19]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[34] GLAWION, Alex. Server vs. Desktop CPUs: What are the differences?. In: *CG Director* [online]. 7 March 2022 [cit. 2022-05-19]. Dostupné z: https://www.cgdirector.com/server-vs-desktop-cpus/

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

[36] ANZT, Hartwig, Tobias RIBIZEL, Goran FLEGAR, Edmond CHOW a Jack DONGARRA. ParILUT - A Parallel Threshold ILU for GPUs. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* [online]. IEEE, 2019, 2019, s. 231-241 [cit. 2022-05-03]. ISBN 978-1-7281-1246-6. Dostupné z: doi:10.1109/IPDPS.2019.00033

[37] SHARMA, Bharatkumar a Jaegeun HAN. *Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++*. Birmingham: Packt Publishing, 2019. ISBN 978-1788996242.

[38] SAAD, Y. *Iterative methods for sparse linear systems*. 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics, 2003. ISBN 978-0898715347.

Appendix A

CUDA matrix multiplication benchmark code

The full code for the matrix multiplication benchmark is shown in Listing A.1 (file `matrixMul.cu`).

```
/**
* Copyright 1993-2015 NVIDIA Corporation.  All rights reserved.
*
* Please refer to the NVIDIA end user license agreement (EULA) associated
* with this source code for terms and conditions that govern your use of
* this software. Any use, reproduction, disclosure, or distribution of
* this software and related documentation outside the terms of the EULA
* is strictly prohibited.
*
*/

/**
* Matrix multiplication: C = A * B.
* Host code.
*
* This sample implements matrix multiplication which makes use of shared memory
* to ensure data reuse, the matrix multiplication is done using tiling
* approach. It has been written for clarity of exposition to illustrate various
* CUDA programming principles, not with the goal of providing the most
* performant generic kernel for matrix multiplication. See also: V. Volkov and
* J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in Proc. 2008
* ACM/IEEE Conf. on Supercomputing (SC '08), Piscataway, NJ: IEEE Press, 2008,
* pp. Art. 31:1-11.
```

```
24 */
25
26 // System includes
27 #include <assert.h>
28 #include <stdio.h>
29
30 // CUDA runtime
31 #include <cuda_runtime.h>
32
33 // Helper functions and utilities to work with CUDA
34 #include <helper_cuda.h>
35 #include <helper_functions.h>
36
37 template <int BLOCK_SIZE>
38 __global__ void MatrixMulCUDAGlobal(float *C, float *A, float *B, int↩
       wA, int wB) {
39   // Each thread computes one element of C by accumulating results ↩
       into Cvalue
40   float Cvalue = 0;
41   int row = blockIdx.y * blockDim.y + threadIdx.y;
42   int col = blockIdx.x * blockDim.x + threadIdx.x;
43   for( int i = 0; i < wA; ++i )
44     Cvalue += A[row * wA + i] * B[i * wB + col];
45
46   C[row * wB + col] = Cvalue;
47 }
48
49 /**
50 * Matrix multiplication (CUDA Kernel) on the device: C = A * B
51 * wA is A's width and wB is B's width
52 */
53 template <int BLOCK_SIZE>
54 __global__ void MatrixMulCUDA(float *C, float *A, float *B, int wA, ↩
      int wB) {
55   // Block index
56   int bx = blockIdx.x;
57   int by = blockIdx.y;
58
59   // Thread index
60   int tx = threadIdx.x;
61   int ty = threadIdx.y;
62
63   // Index of the first sub-matrix of A processed by the block
64   int aBegin = wA * BLOCK_SIZE * by;
65
66   // Index of the last sub-matrix of A processed by the block
67   int aEnd = aBegin + wA - 1;
68
69   // Step size used to iterate through the sub-matrices of A
70   int aStep = BLOCK_SIZE;
71
72   // Index of the first sub-matrix of B processed by the block
73   int bBegin = BLOCK_SIZE * bx;
74
75   // Step size used to iterate through the sub-matrices of B
76   int bStep = BLOCK_SIZE * wB;
```

```
77
78    // Csub is used to store the element of the block sub-matrix
79    // that is computed by the thread
80    float Csub = 0;
81
82    // Loop over all the sub-matrices of A and B
83    // required to compute the block sub-matrix
84    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)↩
          {
85      // Declaration of the shared memory array As used to
86      // store the sub-matrix of A
87      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
88
89      // Declaration of the shared memory array Bs used to
90      // store the sub-matrix of B
91      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
92
93      // Load the matrices from device memory
94      // to shared memory; each thread loads
95      // one element of each matrix
96      As[ty][tx] = A[a + wA * ty + tx];
97      Bs[ty][tx] = B[b + wB * ty + tx];
98
99      // Synchronize to make sure the matrices are loaded
100     __syncthreads();
101
102     // Multiply the two matrices together;
103     // each thread computes one element
104     // of the block sub-matrix
105     #pragma unroll
106
107     for (int k = 0; k < BLOCK_SIZE; ++k) {
108       Csub += As[ty][k] * Bs[k][tx];
109     }
110
111     // Synchronize to make sure that the preceding
112     // computation is done before loading two new
113     // sub-matrices of A and B in the next iteration
114     __syncthreads();
115   }
116
117   // Write the block sub-matrix to device memory;
118   // each thread writes one element
119   int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
120   C[c + wB * ty + tx] = Csub;
121 }
122
123 void ConstantInit(float *data, int size, float val) {
124   for (int i = 0; i < size; ++i) {
125     data[i] = val;
126   }
127 }
128
129 /**
130 * Run a simple test of matrix multiplication using CUDA
131 */
```

```
132 int MatrixMultiply(int argc, char **argv, int block_size, const dim3 ↩
    &dimsA,
133 const dim3 &dimsB) {
134   // Allocate host memory for matrices A and B
135   unsigned int size_A = dimsA.x * dimsA.y;
136   unsigned int mem_size_A = sizeof(float) * size_A;
137   float *h_A;
138   checkCudaErrors(cudaMallocHost((void **)&h_A, mem_size_A));
139   unsigned int size_B = dimsB.x * dimsB.y;
140   unsigned int mem_size_B = sizeof(float) * size_B;
141   float *h_B;
142   checkCudaErrors(cudaMallocHost((void **)&h_B, mem_size_B));
143   cudaStream_t stream;
144
145   // Initialize host memory
146   const float valB = 0.01f;
147   ConstantInit(h_A, size_A, 1.0f);
148   ConstantInit(h_B, size_B, valB);
149
150   // Allocate device memory
151   float *d_A, *d_B, *d_C;
152
153   // Allocate host matrix C
154   dim3 dimsC(dimsB.x, dimsA.y, 1);
155   unsigned int mem_size_C = dimsC.x * dimsC.y * sizeof(float);
156   float *h_C;
157   checkCudaErrors(cudaMallocHost((void **)&h_C, mem_size_C));
158
159   if (h_C == NULL) {
160     fprintf(stderr, "Failed to allocate host matrix C!\n");
161     exit(EXIT_FAILURE);
162   }
163
164   checkCudaErrors(cudaMalloc(reinterpret_cast<void **>(&d_A), ↩
       mem_size_A));
165   checkCudaErrors(cudaMalloc(reinterpret_cast<void **>(&d_B), ↩
       mem_size_B));
166   checkCudaErrors(cudaMalloc(reinterpret_cast<void **>(&d_C), ↩
       mem_size_C));
167   // Allocate CUDA events that we'll use for timing
168   cudaEvent_t start, stop;
169   checkCudaErrors(cudaEventCreate(&start));
170   checkCudaErrors(cudaEventCreate(&stop));
171
172   checkCudaErrors(cudaStreamCreateWithFlags(&stream, ↩
       cudaStreamNonBlocking));
173
174   // copy host memory to device
175   checkCudaErrors(
176   cudaMemcpyAsync(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice, ↩
       stream));
177   checkCudaErrors(
178   cudaMemcpyAsync(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice, ↩
       stream));
179
180   // Setup execution parameters
```

```
181   dim3 threads(block_size, block_size);
182   dim3 grid(dimsB.x / threads.x, dimsA.y / threads.y);
183
184   // Create and start timer
185   printf("Computing result using CUDA Kernel...\n");
186
187   // Performs warmup operation using matrixMul CUDA kernel
188   if (block_size == 16) {
189     MatrixMulCUDA<16>
190     <<<grid, threads, 0, stream>>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
191   } else {
192     MatrixMulCUDA<32>
193     <<<grid, threads, 0, stream>>>(d_C, d_A, d_B, dimsA.x, dimsB.x);
194   }
195
196   printf("done\n");
197   checkCudaErrors(cudaStreamSynchronize(stream));
198
199   // Record the start event
200   checkCudaErrors(cudaEventRecord(start, stream));
201
202   // Execute the kernel
203   int nIter = 10;
204
205   for (int j = 0; j < nIter; j++) {
206     if (block_size == 16) {
207       MatrixMulCUDA<16>
208       <<<grid, threads, 0, stream>>>(d_C, d_A, d_B, dimsA.x, dimsB.x)↩
              ;
209     } else {
210       MatrixMulCUDA<32>
211       <<<grid, threads, 0, stream>>>(d_C, d_A, d_B, dimsA.x, dimsB.x)↩
              ;
212     }
213   }
214
215   // Record the stop event
216   checkCudaErrors(cudaEventRecord(stop, stream));
217
218   // Wait for the stop event to complete
219   checkCudaErrors(cudaEventSynchronize(stop));
220
221   float msecTotal = 0.0f;
222   checkCudaErrors(cudaEventElapsedTime(&msecTotal, start, stop));
223
224   // Compute and print the performance
225   float msecPerMatrixMul = msecTotal / nIter;
226   double flopsPerMatrixMul = 2.0 * static_cast<double>(dimsA.x) *
227   static_cast<double>(dimsA.y) *
228   static_cast<double>(dimsB.x);
229   double gigaFlops =
230   (flopsPerMatrixMul * 1.0e-9f) / (msecPerMatrixMul / 1000.0f);
231   printf(
232   "Performance= %.2f GFlop/s, Time= %.3f msec, Size= %.0f Ops,"
233   " WorkgroupSize= %u threads/block\n",
```

```
234   gigaFlops, msecPerMatrixMul, flopsPerMatrixMul, threads.x * ↩
          threads.y);
235
236   // Copy result from device to host
237   checkCudaErrors(
238   cudaMemcpyAsync(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost, ↩
          stream));
239   checkCudaErrors(cudaStreamSynchronize(stream));
240
241   printf("Checking computed result for correctness: ");
242   bool correct = true;
243
244   // test relative error by the formula
245   //     |<x, y>_cpu - <x,y>_gpu|/<|x|, |y|>  < eps
246   double eps = 1.e-6;  // machine zero
247
248   for (int i = 0; i < static_cast<int>(dimsC.x * dimsC.y); i++) {
249     double abs_err = fabs(h_C[i] - (dimsA.x * valB));
250     double dot_length = dimsA.x;
251     double abs_val = fabs(h_C[i]);
252     double rel_err = abs_err / abs_val / dot_length;
253
254     if (rel_err > eps) {
255       printf("Error! Matrix[%05d]=%.8f, ref=%.8f error term is > %E\n↩
              ", i,
256       h_C[i], dimsA.x * valB, eps);
257       correct = false;
258     }
259   }
260
261   printf("%s\n", correct ? "Result = PASS" : "Result = FAIL");
262
263   // Clean up memory
264   checkCudaErrors(cudaFreeHost(h_A));
265   checkCudaErrors(cudaFreeHost(h_B));
266   checkCudaErrors(cudaFreeHost(h_C));
267   checkCudaErrors(cudaFree(d_A));
268   checkCudaErrors(cudaFree(d_B));
269   checkCudaErrors(cudaFree(d_C));
270   checkCudaErrors(cudaEventDestroy(start));
271   checkCudaErrors(cudaEventDestroy(stop));
272   printf(
273   "\nNOTE: The CUDA Samples are not meant for performance"
274   "measurements. Results may vary when GPU Boost is enabled.\n");
275
276   if (correct) {
277     return EXIT_SUCCESS;
278   } else {
279     return EXIT_FAILURE;
280   }
281 }
282
283 /**
284 * Program main
285 */
286 int main(int argc, char **argv) {
```

```
287   printf("[Matrix Multiply Using CUDA] - Starting...\n");
288
289   if (checkCmdLineFlag(argc, (const char **)argv, "help") ||
290   checkCmdLineFlag(argc, (const char **)argv, "?")) {
291     printf("Usage -device=n (n >= 0 for deviceID)\n");
292     printf("      -wA=WidthA -hA=HeightA (Width x Height of Matrix A)↩
          \n");
293     printf("      -wB=WidthB -hB=HeightB (Width x Height of Matrix B)↩
          \n");
294     printf(
295     "  Note: Outer matrix dimensions of A & B matrices"
296     " must be equal.\n");
297
298     exit(EXIT_SUCCESS);
299   }
300
301   // This will pick the best possible CUDA capable device, otherwise
302   // override the device ID based on input provided at the command ↩
       line
303   int dev = findCudaDevice(argc, (const char **)argv);
304
305   int block_size = 32;
306
307   int mul = 10;
308
309   dim3 dimsA(mul * block_size, mul * block_size, 1);
310   dim3 dimsB(mul * block_size, mul * block_size, 1);
311
312   // width of Matrix A
313   if (checkCmdLineFlag(argc, (const char **)argv, "wA")) {
314     dimsA.x = getCmdLineArgumentInt(argc, (const char **)argv, "wA");
315   }
316
317   // height of Matrix A
318   if (checkCmdLineFlag(argc, (const char **)argv, "hA")) {
319     dimsA.y = getCmdLineArgumentInt(argc, (const char **)argv, "hA");
320   }
321
322   // width of Matrix B
323   if (checkCmdLineFlag(argc, (const char **)argv, "wB")) {
324     dimsB.x = getCmdLineArgumentInt(argc, (const char **)argv, "wB");
325   }
326
327   // height of Matrix B
328   if (checkCmdLineFlag(argc, (const char **)argv, "hB")) {
329     dimsB.y = getCmdLineArgumentInt(argc, (const char **)argv, "hB");
330   }
331
332   if (dimsA.x != dimsB.y) {
333     printf("Error: outer matrix dimensions must be equal. (%d != %d)\↩
          n",
334     dimsA.x, dimsB.y);
335     exit(EXIT_FAILURE);
336   }
337
```

```
338   printf("MatrixA(%d,%d), MatrixB(%d,%d)\n", dimsA.x, dimsA.y, ↩
          dimsB.x,
339   dimsB.y);
340
341   int matrix_result = MatrixMultiply(argc, argv, block_size, dimsA, ↩
          dimsB);
342
343   exit(matrix_result);
344 }
```

Listing A.1: Matrix multiplication benchmark code. Taken from Nvidia's samples located in the users home directory by default: `$HOME/NVIDIA-samples/0_Introduction/matrixMul/`.