České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Obor: Aplikace softwarového inženýrství

Paralelní výpočet LU rozkladu na GPU pro numerické řešení parciálních diferenciálních rovnic

Parallel Computation of LU Decomposition on GPUs for the Numerical Solution of Partial Differential Equations

DIPLOMOVÁ PRÁCE

Vypracoval: Bc. Lukáš Čejka
Vedoucí práce: doc. Ing. Tomáš Oberhuber, Ph.D.
Rok: 2023

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství

Akademický rok 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Lukáš Matthew Čejka
Studijní program:	Aplikace přírodních věd
Obor:	Aplikace softwarového inženýrství
Název práce česky:	Paralelní výpočet LU rozkladu na GPU pro numerické řešení parciálních diferenciálních rovnic
Název práce anglicky:	Parallel Computation of LU Decomposition on GPUs for the Numerical Solution of Partial Differential Equations
Jazyk práce:	Angličtina

Pokyny pro vypracování:

1. Implementujte tzv. pivoting při výpočtu LU rozkladu.
2. Aplikujte paralelní výpočet LU rozkladu pro inverzi Schurova doplňku v metodě BDDC.
3. Porovnejte efektivitu výsledné implementace s některými knihovnami pro výpočet LU rozkladu na CPU i GPU.
4. Proveďte výpočetní studii a porovnejte efekt pivotingu při řešení této úlohy.

Doporučená literatura:

[1] DONGARRA, J., GATES, M., HAIDAR, A., KURZAK, J., LUSZCZEK, P., WU, P., YAMAZAKI, I., YARKHAN, A., ABALENKOVS, M., BAGHERPOUR, N., HAMMARLING, S., ŠÍSTEK, J., STEVENS, D., ZOUNON, M. a RELTON, S. D. PLASMA. *ACM Transactions on Mathematical Software*. 2019. Vol. 45, no. 2p. 1-35. DOI 10.1145/3264491.

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

[3] ANZT, H., RIBIZEL, T., FLEGAR, G., CHOW, E. a DONGARRA, J. ParILUT - A Parallel Threshold ILU for GPUs. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019. p. 231-241. ISBN 978-1-7281-1246-6.

Jméno a pracoviště vedoucího práce:

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

.....
vedoucí práce

Datum zadání diplomové práce: 12. 10. 2022

Termín odevzdání diplomové práce: 3. 5. 2023

Doba platnosti zadání je dva roky od data zadání.

.....
garant oboru

.....
vedoucí katedry

.....
děkan

V Praze dne 12. 10. 2022

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

Declaration

I declare that I have carried out my Master's Degree Project independently and I have used only the materials (literature, projects, software, etc.) listed in the bibliography.

V Praze dne ....

Bc. Lukáš Čejka

Poděkování

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

Acknowledgment

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

Bc. Lukáš Čejka

Název práce:
Paralelní výpočet LU rozkladu na GPU pro numerické řešení parciálních diferenciálních rovnic

Autor: Bc. Lukáš Čejka

Studijní program: Aplikace přírodních věd
Obor: Aplikace softwarového inženýrství
Druh práce: Diplomová práce

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

Abstrakt: TODO

Klíčová slova: TODO

Title:
Parallel Computation of LU Decomposition on GPUs for the Numerical Solution of Partial Differential Equations

Author: Bc. Lukáš Čejka

Abstract: TODO

Key words: TODO

Contents

Nomenclature

Abbreviation	Meaning
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SIMT	Single-Instruction Multiple-Thread
SIMD	Single-Instruction Multiple-Data
CM	Crout Method
CMPP	Crout Method with Partial Pivoting
ICMPP	Iterative Crout Method with Partial Pivoting
PCM	Parallel Crout Method
PCMx	Parallel Crout Method that uses 1D thread blocks of x threads
ICM	Iterative Crout Method
ICMx	Iterative Crout Method that uses 2D thread blocks of x by x threads
IS	Iterative Solver
ISx	Iterative Solver that uses 1D thread blocks of x threads
HPC	High-Performance Computing
LUP	Lower-Upper decomposition with partial Pivoting

Table 1: Abbreviations used in this project.

Introduction

TODO

Chapter 1

Theory

This chapter will introduce the theory behind the core parts of this project. First, Nvidia's Graphics Processing Units (GPUs) and their use in General-Purpose Computing (GPGPU) will be briefly described. Then, CUDA, the API for communicating with Nvidia GPUs, will be presented. The final part introduces the method implemented in this project: Iterative Crout's Method with partial pivoting (ICMPP).

1.1 GPUs

In essence, a Graphics Processing Unit (GPU) is a device purpose-built for accelerating the graphical output of a computer system. Simply put, a GPU comprises many smaller processing units. While these units are not as fast as the cores of a CPU, they excel at processing many similar tasks in parallel. An example of such a task is displaying an image on a monitor. In simple terms, to display the image, each pixel must be computed and rendered. If the image is 1,200 by 1,200 pixels, then, the processing entity must compute and render 1,440,000 individual points. Given the many processing units that the GPU has, it is capable of performing this operation efficiently.

While the common understanding of a GPU is that it is an image-processing device, how it processes images can be leveraged for non-graphical computation-heavy tasks. This is referred to as General-Purpose Computing on Graphics Processing Units (GPGPU).

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

General-Purpose Computing on Graphics Processing Units (GPGPU) has become an integral part of High-Performance Computing (HPC) in the last two decades [1, 2]. One of the key parts of GPGPU is to tailor tasks for efficient execution on GPUs. To use a GPU efficiently, it is necessary to parallelize the workload. A common example of a parallelizable task is the multiplication of two matrices. Without parallelism and assuming the naive matrix-multiplication algorithm, this task requires the processing entity to perform the pseudocode shown in Listing 1.1.

```
multiplyMatrices(A, B, C):
  for i from 0 to m - 1:
    for j from 0 to p - 1:
      for k from 0 to n - 1:
        C[i, j] += A[i, k] * B[k, j]
```

Listing 1.1: Pseudocode for the multiplication of two matrices. Let **A** be a $m \times n$ matrix, **B** a $n \times p$ matrix, and **C** a $m \times p$ matrix.

Since parallelism is not used, the computation is strictly sequential and would arguably benefit from the faster core clock speeds of a CPU [2]. However, since the computation of every element in **C** is independent of the other elements, it is possible to compute all elements of **C** simultaneously. In other words, every thread of the GPU can be tasked to compute one element of **C**, i.e. each thread would only compute the innermost loop.
Note that the parallelization of some tasks is not as straightforward, for example, Crout Method with partial pivoting (detailed in Section 1.3.2).

Similarly to the above-mentioned example of displaying the image on a monitor, the GPU - and its many processing units - greatly benefits from the parallelized workload largely due to its architecture. To illustrate this, Figure 1.1 shows a general architecture comparison of a CPU and a GPU.

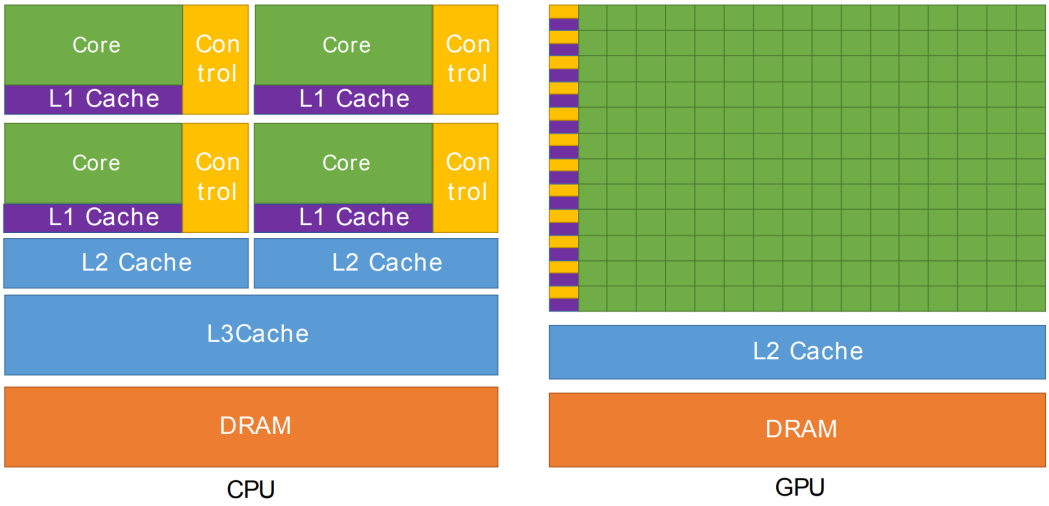

Figure 1.1: Architecture comparison of a CPU and a GPU. Taken from *CUDA C++ Programming Guide* [3].

In Figure 1.1 it can be seen that the control units and caches of the CPU are proportionately sized to illustrate that GPUs do not focus on fine-grained cache control and prediction logic as much as CPUs. The reasoning behind this is that CPUs in general attempt to use the aforementioned focus points to hide memory latency. On the other hand, GPUs do not attempt to hide memory latency with controllers using costly caching. Instead, as mentioned in *CUDA C++ Best Practices Guide* [4], it is considered best practice for the developer to hide memory latency using CUDA-specific functionalities [5].

In Nvidia GPUs, the control units are referred to as Stream Multiprocessors (SMs) and they are responsible for scheduling tasks to processing units. Therefore, it can be argued that the computational power of an Nvidia GPU depends on - among other factors - the number of SMs it has and their capabilities. The reasoning behind this is that the more SMs a GPU has, the more concurrent computations it is capable of performing. To exemplify this, see Table 1.1 for a selection of the latest Nvidia Datacenter cards and their relevant specifications.

The industry-standard values used to compare GPUs are TFLOPS and memory bandwidth as both are crucial for any computation. From Table 1.1 it can be seen that the H100 is capable of

GPU Features	V100	A100	H100
GPU	GV100 (Volta)	GA100 (Ampere)	GH100 (Hopper)
GPU Board Form Factor	SXM2	SXM4	SXM5
SMs	80	108	132
FP32 Cores / SM	64	64	128
FP32 Cores / GPU	5,120	6,912	16,896
FP64 Cores / SM	32	32	64
FP64 Cores / GPU	2,560	3,456	8,448
GPU Boost Clock	1,530 MHz	1,410 MHz	1,830 MHz
Peak FP32 TFLOPS	15.7	19.5	66.9
Peak FP64 TFLOPS	7.8	9.7	33.5
Memory Interface	4,096-bit HBM2	5,120-bit HBM2	5,120-bit HBM3
Memory Size	32 GB	80 GB	80 GB
Memory Bandwidth	900 GB/s	2,039 GB/s	3,352 GB/s
L2 Cache Size	6 MB	40 MB	50 MB
Max. Shared Memory Size / SM	96 KB	164 KB	228 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	20,480 KB	27,648 KB	33,792 KB
TDP	300 Watts	400 Watts	700 Watts
Transistors	21.1 billion	54.2 billion	80 billion

Table 1.1: Comparison of GPUs: V100 (Volta architecture; released in December 2017), A100 (Ampere architecture; released in May 2020), H100 (Hopper architecture; released in September 2022). FP stands for Floating Point; TFLOPS signifies the number of trillion floating-point operations the processor can perform per second; TDP stands for Thermal Design Power which can be used as an indicator of power consumption under the maximum theoretical load [6]. The specifications of each GPU in this table are the best possible version of the card available as of February 2023, i.e. SXM instead of PCIe and the version with the most VRAM available. Interesting aspects are highlighted in green. The data was obtained from various sources for the V100 [7], the A100 [8, 9], and the H100 [10].

performing up to 3.4 times as many FP operations as its predecessor, the A100. Similarly, the H100 outperforms the A100 by a factor of 1.6 when it comes to memory bandwidth. Unfortunately, even though the H100 was unveiled in April 2022 and released in September 2022 there is a global shortage as of March 2023[1].

To fully utilize the computational power of its GPUs, Nvidia provides developers with its API, CUDA.

1.2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) was introduced by Nvidia in 2006 as a programming model; however, it is also often referred to as a parallel computing platform [11]. Its primary objective is to provide developers with low-level access to GPU hardware, including, but not limited to, the ability to fine-tune the allocation of processing units and memory. This enables developers to fully utilize a GPU's potential and customize its use for specific applications. CUDA supports the

[1]Update: 'Huge' GPU supply shortage due to AI needs, analyst Dylan Patel says URL: `https://www.fierceelectronics.com/electronics/ask-nvidia-ceo-gtc-gpu-shortage-looming`

following programming languages: C, C++, and Fortran. Additionally, wrappers by third parties allow it to be used in other languages, such as Python, Perl, Java, Ruby, MATLAB, and Julia [12].

First, a selection of fundamental CUDA-related terms used in this project will be presented. Then, CUDA's thread management system will be briefly described. After that, the CUDA memory management system will be introduced. The last but one part will describe the concurrent execution of code on the GPU, and finally, a parallel computing concept used in this project will be presented. Along with the explanation of the topic, each section can contain examples of C++ CUDA extensions as C++ was used for the development of this project.

1.2.1 Fundamental Terminology

For a better understanding of CUDA-related concepts, it is necessary to introduce fundamental terminology [13, 2]:

- *Host* - CPU and its memory. The host provides the GPU with instructions to execute, for example, transferring data, executing code, synchronizing the GPU, etc.
- *Device* - GPU and its memory. The device executes instructions issued by the host.
- *Kernel* - C++ function that is called from the host and executed on the device based on a configuration. In C++, the definition of a kernel is prefixed using the `__global__` keyword.

1.2.2 Thread Management

This section aims to briefly introduce the thread management system present in CUDA - for a detailed explanation see *Formats for storage of sparse matrices on GPU* [1] and *Parallel LU Decomposition for the GPU* [2]. To fully leverage the performance Nvidia GPUs can provide, it is paramount to manage their execution units well. For this purpose, CUDA provides many functionalities and structures [3].

CUDA thread According to *CUDA C++ Programming Guide* [3], a CUDA thread is "*an executed sequence of operations*". It represents the most fundamental level of abstraction for carrying out computations or memory operations. It is lightweight in design which allows the GPU to switch between threads seamlessly. Note that in the context of this project, a CUDA thread may also be referred to simply as a *thread*. In CUDA, threads are organized into hierarchical groups, with the *warp* being the most fundamental group.

Warp CUDA uses the Single-Instruction Multiple-Thread (SIMT) architecture. As the name suggests, in SIMT, threads execute the same instruction in parallel with the added possibility of each thread using different data [14]. At the most elementary level of the thread-group hierarchy, threads execute in lockstep as a group of 32 called a *warp*. Note that a warp can only comprise consecutive threads. For a clearer understanding, the execution of instructions by an 8-thread warp is shown in Figure 1.2.

While threads in a warp execute in lockstep, they can diverge and execute different instructions. This behavior is referred to as *thread divergence* and it is discouraged as the execution will be serialized and thus suboptimal - see Figure 1.3 for a visualization of thread divergence. Interestingly,

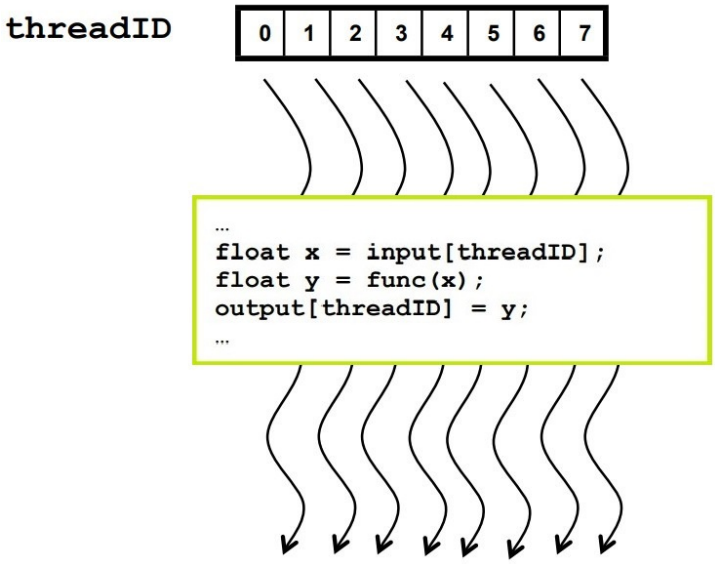

Figure 1.2: Execution of code by an 8-thread warp. The unique ID of each thread is stored in the variable `threadID`. In this example, `threadID` is used to read and write different data. Taken from *Getting Started with CUDA* [13] and *Parallel LU Decomposition for the GPU* [2].

until Nvidia's Volta generation was introduced in December 2017, thread divergence could lead to a deadlock in specific cases where sharing data between threads of a warp was required [3, 15].

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

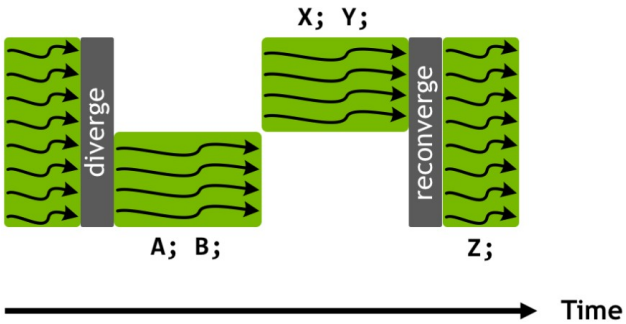

Figure 1.3: Pseudocode showcasing thread divergence in an 8-thread warp. The `threadIdx.x` variable contains each thread's ID in the 1st dimension. In this example, while threads `0` to `3` execute statements `A` and `B`, threads `4` to `7` are idle. Then, threads `4` to `7` execute statements `X` and `Y` while threads `0` to `3` are idle. Finally, all threads resume lockstep execution with statement `Z`. Taken from Nvidia's developer blog post: *Inside Volta: The World's Most Advanced Data Center GPU* [16].

While CUDA provides special functions for intra-warp thread management [17], their use is not common compared to the use of thread groups higher in the hierarchy, called *blocks*.

Block To execute a group of threads they must first be grouped into a *block*. A single block can comprise up to 1024 threads (as of CUDA compute capability 3.5) that are organized in either 1D, 2D, or 3D [3]. Every thread of a particular block has a unique ID per dimension, i.e. `x` for 1D, `x`, `y` for 2D, and `x`, `y`, `z` for 3D. Combined with the fact that the threads of a warp are consecutive, choosing the dimension and size of a thread block can play a crucial role in achieving optimal

performance. To illustrate this, an example showing threads of a block divided into warps can be seen in Figure 1.4.

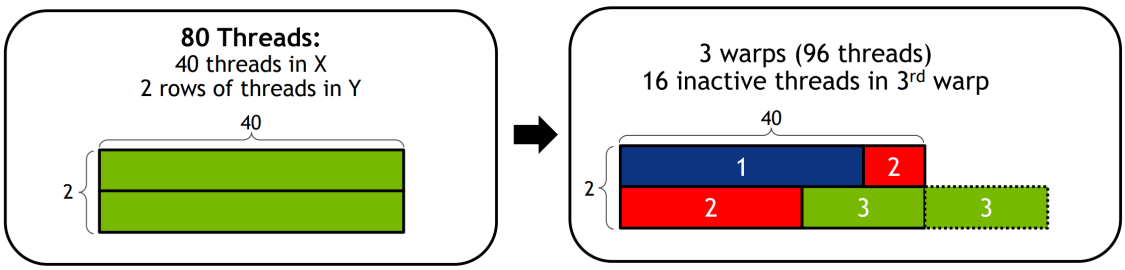

Figure 1.4: Division of threads into warps in a block of 40 by 2 threads. Each green rectangle in the left image represents a row of threads. In the right image, each color represents a unique warp. Taken from *VOLTA Architecture and performance optimization* [18].

As can be seen in Figure 1.4, since threads are consecutive in their 1st dimension, the 1st row of the block comprises 32 threads that belong to warp `0` (blue rectangle) and 8 threads from warp `1` (upper-right red square). The 2nd row comprises 24 threads belonging to warp `1` (lower-right red rectangle) and 16 threads from warp `2` (left-most green rectangle). While a warp can only consist of 32 consecutive threads, in warp `2`, only 16 will be utilized for execution, while the remaining 16 will remain inactive.

Grid The final group of threads in the hierarchy, the *grid*, comprises thread blocks. Similarly to how threads are structured within a thread block, blocks can be structured within grids. Grids can consist of up to 3 dimensions of blocks; each block has a unique ID per dimension. Additionally, the limit of thread blocks per grid is bound to each dimension: the maximum number of blocks per dimension is 65,536. For a visualization of a 2D grid comprising 2D thread blocks see Figure 1.5.

As shown in Figures 1.2 and 1.3, within a kernel, each thread has access to a collection of predefined variables unique to it. The following is a selection of such commonly-used variables:

- `threadIdx` - a 3-component vector containing the IDs of a thread in each dimension of a thread block. Specifically, `threadIdx.x` contains the thread's ID in the 1st dimension (`x`), `threadIdx.y` in the 2nd dimension (`y`), and `threadIdx.z` in the 3rd dimension (`z`). For example, in the 3-by-4 thread block shown in Figure 1.5, the values of `threadIdx` for the bottom-right-most thread are `{3, 2, 1}`, i.e., `threadIdx.x` = 3, `threadIdx.x` = 2, `threadIdx.z` = 1.
- `blockIdx` - a 3-component vector containing the IDs of a block in each dimension of a grid. Similarly to `threadIdx`, each vector component contains the block's ID in a specific dimension, i.e., `blockIdx.x` contains the ID of the block in the 1st dimension (`x`), etc. For example, for the bottom-right-most thread block shown in Figure 1.5, `blockIdx` would be `{2, 1, 1}`, i.e., `blockIdx.x` = 2, `blockIdx.y` = 1, `blockIdx.z` = 1.
- `blockDim` - a 3-component vector containing the sizes of a block's dimensions. For example, for the thread block shown in Figure 1.5, `blockDim` would be `{3, 4, 1}`, i.e., `blockDim.x` = 3, `blockDim.y` = 4, `blockDim.z` = 1.

For a summarized overview of predefined variables and functions available, see *CUDA syntax* [19] or see *CUDA C++ Programming Guide* [3] for an extensive overview.

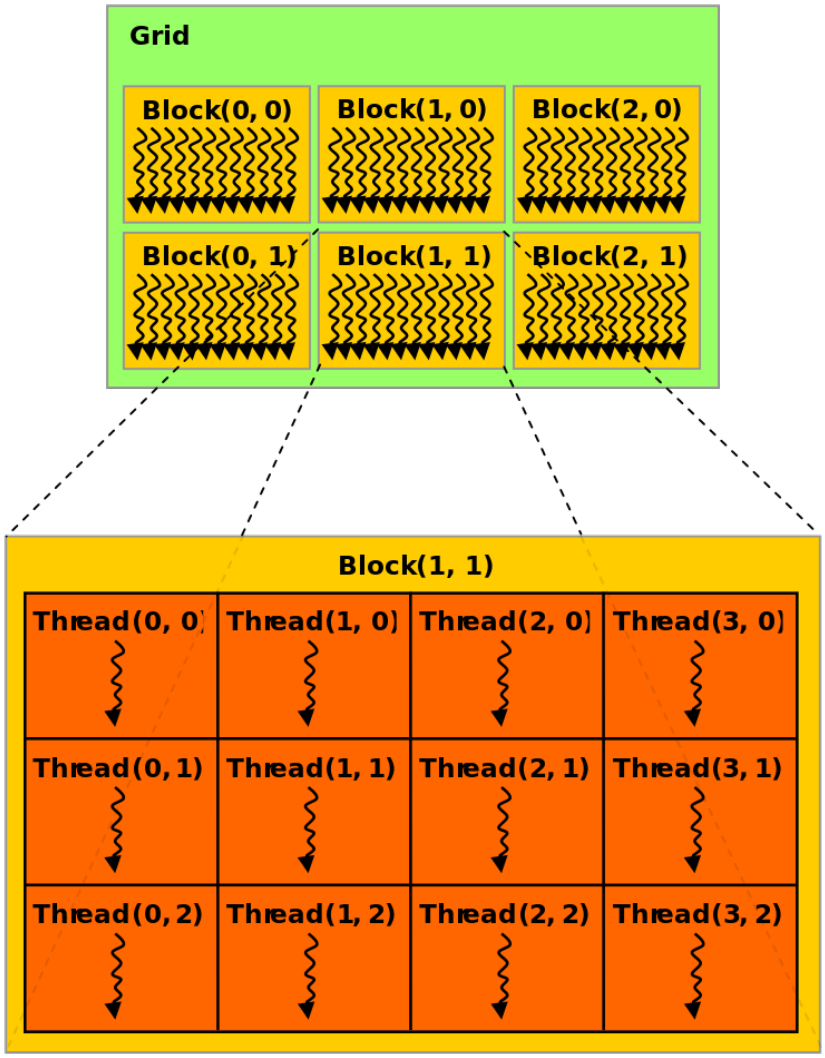

Figure 1.5: Visualization of a 2-by-3 grid comprising 3-by-4 thread blocks. The numbers in the brackets are the IDs of each entity in each dimension. Taken from *CUDA C++ Programming Guide* [3].

The variables listed earlier are often used to compute the global ID of a thread in a dimension of a grid:

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

In a standard setup, the grid is the upper-most thread grouping in CUDA's thread management hierarchy. As visualized in Figure 1.6, when a kernel is executed on the device, it is executed on all threads of a grid.

To execute a kernel on the device, it is necessary to provide the kernel with an execution configuration: `<<< Dg, Db, Ns, S >>>`, where:

- `Dg` - dimension and size of the grid;
- `Db` - dimension and size of each block;

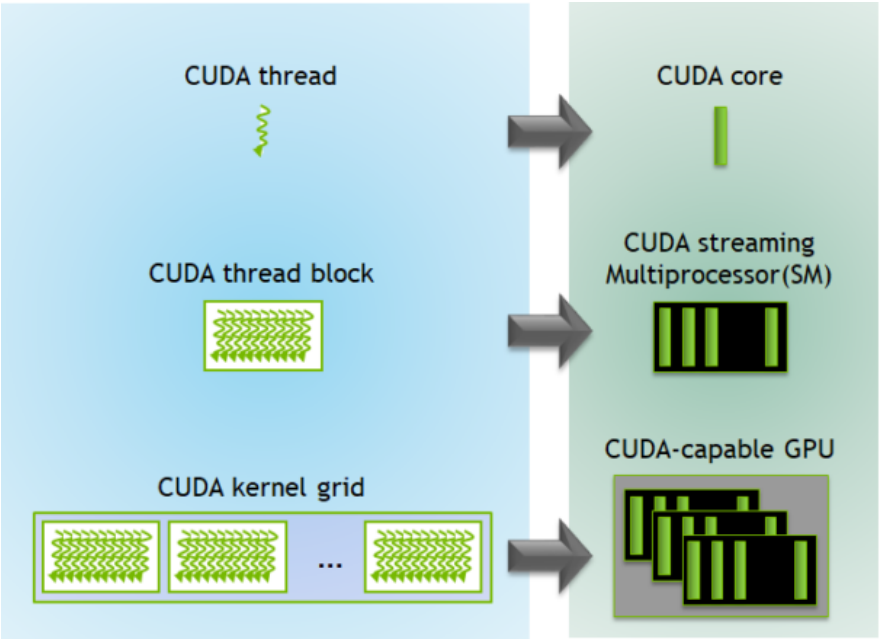

Figure 1.6: Visualization illustrating the execution of CUDA thread structures by different hardware components in an Nvidia GPU. Taken from *CUDA Refresher: The CUDA Programming Model* [20].

- `Ns` - optional (default value is 0), number of bytes in shared memory allocated per block in addition to statically allocated shared memory (explained in Section 1.2.3);
- `S` - optional (default value is 0), stream associated with the kernel (explained in Section 1.2.4).

An example of a kernel call with a basic execution configuration is presented in Listing 1.2.

```
1  // Kernel definition
2  __global__ void AddOneToMatrix( float mtx[N][N] )
3  {
4    // Get thread ID for each dimension to use as matrix indices
5    int row = blockIdx.x * blockDim.x + threadIdx.x;
6    int col = blockIdx.y * blockDim.y + threadIdx.y;
7
8    // Check if the indices are within the dimensions of the matrix
9    if( row < N && col < N ) {
10     mtx[row][col] += 1;
11   }
12 }
13
14 int main()
15 {
16   ...
17   // Number of threads in each 2D block is 16x16 = 256
18   dim3 threadsPerBlock( 16, 16 );
19
20   // Number of blocks in the grid depends on the matrix dimension (NxN)
21   dim3 numBlocks( N / threadsPerBlock.x, N / threadsPerBlock.y );
22
23   // Kernel that computes: mtx = mtx + 1 (element-wise)
24   AddOneToMatrix<<< numBlocks, threadsPerBlock >>>( mtx );
25   ...
26 }
```

Listing 1.2: Excerpt of C++ code portraying the creation of a grid configuration and the launching of a kernel. The example implemented aims to increment every value of the `N`-by-`N` matrix `mtx` by 1 using the device. `dim3` is an integer vector type based on `uint3` that is used to specify dimensions - unspecified components are implicitly set to 1 [3]. Taken from *CUDA C++ Programming Guide* [3].

In Listing 1.2, the code in the `main()` function is executed on the host while the code in the `AddOneToMatrix(...)` function is executed on the device. From the host's point of view, kernel calls are asynchronous. In other words, the host will call for the kernel to be executed on the device and then it will move on to the next line of execution without delay. This holds even if the host calls two kernels in succession. In such a case, from the device's point of view, the two kernels will be executed in the order they were called.

1.2.3 Memory Management

This section will briefly introduce the memory management system present in CUDA - for a comprehensive overview see *Formats for storage of sparse matrices on GPU* [1] and *Parallel LU Decomposition for the GPU* [2].

Similarly to the layout in Section 1.2.2, the memory management system of CUDA will be briefly described starting with the most fundamental type of memory and advancing to the highest level.

Registers and local memory While executing a kernel each thread has access to a memory space that is unique to it, i.e., other threads cannot access it. The lifetime of this memory is bound to the context of the kernel it is allocated with. This space encompasses two separate memories: *registers* and *local memory*.

Registers are fast 32-bit on-chip memories (low latency and high bandwidth) that are often used for storing variables unique to each thread. The number of registers available to a thread in the context of a kernel is limited to 255 (since CUDA compute capability 3.2) [3]. If a variable or structure allocated in the context of a kernel exceeds the registers available to a thread, then the compiler stores them in *local memory* - this behavior is referred to as *register spilling*.

Local memory resides in off-chip device memory (referred to as *global memory*; high latency and low bandwidth) which is slower to access than registers. Local memory for each thread is limited to 512 KB [3].

While using registers to store frequently-accessed indexing variables can help improve performance, it should be done with care to avoid the performance loss associated with register spilling.

Shared memory In the context of a kernel, all threads of a block have access to a block-unique memory space referred to as *shared memory*. Similarly to *registers*, shared memory is located on-chip and is therefore high-speed. The maximum size of shared memory per block varies depending on the CUDA compute capability version as shown in Table 1.2.

Shared memory can be allocated statically, dynamically, or both. The amount of statically allocated memory is known at compile time while the amount of dynamically allocated memory is known at run time [21]. Note that the total amount of shared memory allocated for a thread block is equal to the sum of statically and dynamically allocated shared memory.

Compute capability	7.0 - 7.2	7.5	8.0	8.6	8.7	8.9	9.0
Max. shared memory per block [KB]	96	64	163	99	163	99	227

Table 1.2: Maximum shared memory per thread block depending on the CUDA compute capability version. Note that to use more than 48 KB dynamic shared memory must be used. Taken from *CUDA C++ Programming Guide* [3].

Since *statically* allocated shared memory is known at compile time, it must be declared with a size known at compile time. It is often declared inside a kernel using the `__shared__` modifier.

On the other hand, the size of *dynamically* allocated memory must be supplied as one of the kernel launch parameters as mentioned in Section 1.2.2.

Listing 1.3 shows an excerpt of C++ code where the syntax for static and dynamic memory allocation is presented - for the full code see *Using Shared Memory in CUDA C/C++* by Harris, M. [21].

```
__global__ void kernel( ... )
{
  ...
  // Declare statically allocated shared memory
  __shared__ int s_s[32];

  // Access dynamically allocated shared memory
  extern __shared__ int s_d[];
  ...
}

int main()
{
  ...
  const int n = 64;

  // Launch kernel with:
  // - 32*sizeof(int) bytes of statically allocated shared memory
  // - n*sizeof(int) bytes of dynamically allocated shared memory
  kernel<<< 1, n, n*sizeof(int) >>>( ... );
  ...
}
```

Listing 1.3: Excerpt of C++ code showcasing the syntax of static and dynamic shared memory allocation. Note when an array in shared memory is declared using `extern`, then the size of the array is determined at run time [3]. Taken from *Using Shared Memory in CUDA C/C++* by Harris, M. [21].

For more information regarding shared memory see Section 1.2.3 in *Parallel LU Decomposition for the GPU* [2] or *CUDA C++ Programming Guide* [3].

Global memory The largest memory space present in CUDA is *global memory*. It is occasionally referred to as *device memory* as it resides in an Nvidia GPU's DRAM (Dynamic Random Access Memory). While global memory resides in a device's memory it can be accessed by both the host and the device, thus, creating a communication medium between the two. The amount of global memory available depends on the GPU used, for example, the Nvidia A100 GPU is available with

either 40 GB or 80 GB (shown in Table 1.1). Although it is the largest memory space found on the device, it is high-latency and low-bandwidth compared to, e.g., shared memory.

To clarify, since global memory is accessible by the device, it can be accessed by any thread of any grid. See Figure 1.7 for a visualization of CUDA thread structures and the memory spaces accessible to them.

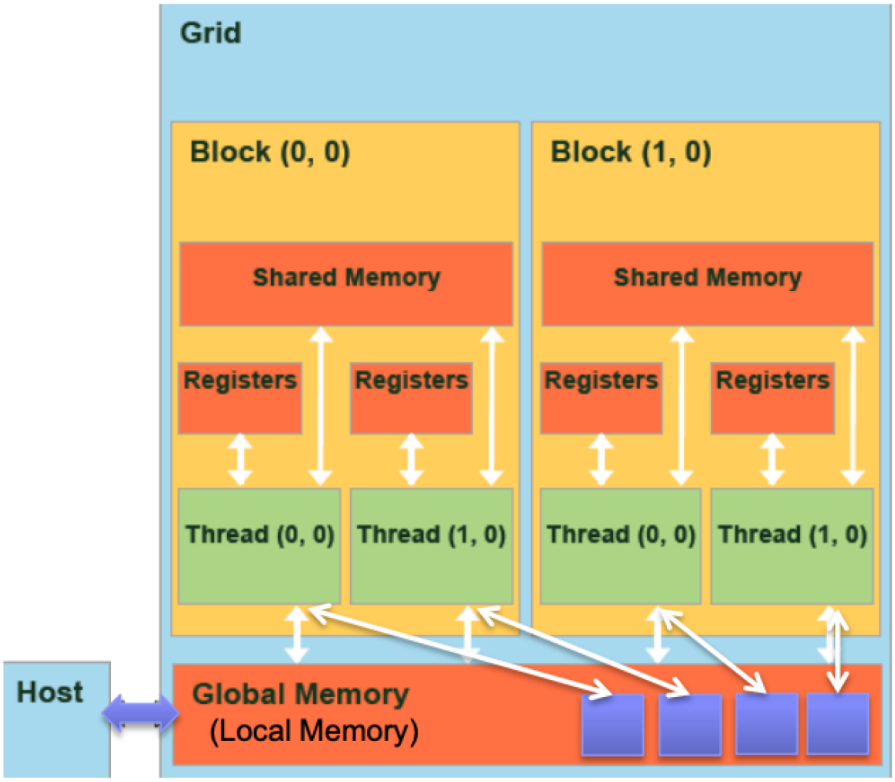

Figure 1.7: CUDA memory structuring with accesses available to each thread structure. Taken from *GPU* by Hsiao, Y. [22].

Furthermore, unlike the aforementioned memory spaces, the lifetime of global memory is bound to the CUDA application, i.e., data can be present in global memory when the application starts and removed when the application terminates. Moreover, global memory requires explicit allocation, copying, and deallocation of data - see *Parallel LU Decomposition for the GPU* [2] and *CUDA C++ Programming Guide* [3] for the specific functions.

For completeness, Figure 1.8 shows the architecture of an Nvidia GPU along with an overview of the CUDA programming model.

While there is only one kernel present in Figure 1.8, it is noteworthy that if another kernel was launched concurrently (explained later in Section 1.2.4) the threads allocated to it would also have access to global memory. In other words, global memory is the same for all kernels in a CUDA application.

1.2.4 Concurrent Kernel Execution

Concurrent kernel execution describes a scenario where two or more kernels from the same CUDA application are running simultaneously. Note that kernels can be executed concurrently only on certain devices with compute capability 2.x or higher. To check whether a device is capable of

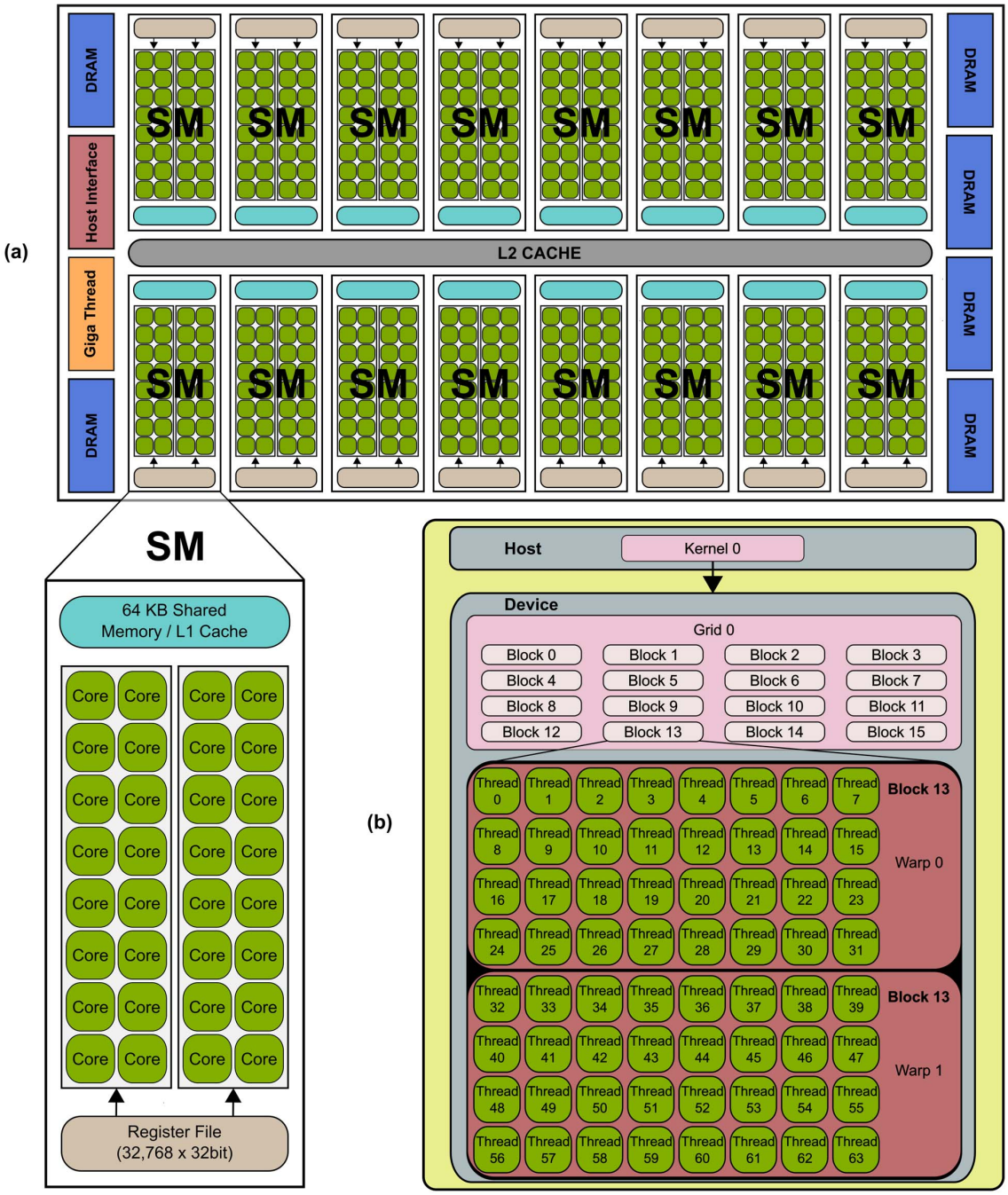

Figure 1.8: Simplified overview of an Nvidia GPU's architecture and the CUDA programming model. Taken from *Correction* [23].

concurrent kernel execution the `deviceQuery` executable provided with the CUDA Toolkit can be used [3].

Streams As mentioned in Section 1.2.2, tasks on the device are executed serially, i.e., in the order the host launched them. This series of tasks is referred to as the *default stream* since it is present implicitly. To execute kernels concurrently, it is necessary to use multiple *streams*. Nvidia defines a stream as "*a sequence of commands that execute in order.*" [3].

Additionally, in Section 1.2.2 it was mentioned that a kernel is launched on a grid made up of thread blocks. This holds even when multiple kernels are launched simultaneously, i.e., each kernel is launched on a separate grid. When a grid is executing a kernel it is referred to as a resident grid.

Note that the use of concurrent kernel execution is associated with certain limitations [2]:

1. Maximum number of resident grids per device.
2. Kernels from different CUDA contexts cannot be run concurrently.
3. Kernels are only run concurrently if the device has enough resources for them.
4. Undefined behavior during stream interaction.

The first limitation is dependent on the compute capability version as presented in Table 1.3.

Compute capability	7.0	7.2	7.5 - 9.0
Max. resident grids per device	128	16	128

Table 1.3: Maximum number of resident grids on one device for CUDA compute capability 7.0 and higher. Taken from *CUDA C++ Programming Guide* [3].

The second limitation prevents the concurrent execution of kernels from two distinct CUDA applications on the same device.

The third limitation states that a device can only run kernels concurrently if its resource limits are not overstepped. However, this does not mean that the kernels will not be run. On the contrary, if two or more kernels are launched using unique streams and the device's resources cannot satisfy the sum of resources required by the kernels, then one of the kernels will be executed first while the other kernels are put aside until resources are available for them. In this context, resources are either local memory or the number of threads the device is capable of running at a point in time.

The fourth limitation is often issued as a disclaimer from Nvidia that warns against the interactivity of streams. Specifically, since all streams are independent of each other and have access to the same global memory, one stream can alter the data used by another stream resulting in inconsistent and possibly erroneous behavior.

To prevent the unpredictable execution order of stream-unique kernels, explicit synchronization can be used. Functions providing explicit synchronization are, for example:

- `cudaDeviceSynchronize()` - this function creates a checkpoint where the host waits until all preceding commands in all streams of all host threads have been completed [3].
- `cudaStreamSynchronize( stream )` - this function serves a similar purpose to `cudaDeviceSynchronize()` with the exception that the host only waits until the preceding commands of a specific stream have completed.

To use a stream, other than the default stream, it is first necessary to construct it by instantiating and creating a stream object using `cudaStream_t` and `cudaStreamCreate()`. Then, the stream must be supplied to a kernel's execution configuration to launch the kernel using it. Once the stream is no longer needed it must be destroyed using `cudaStreamDestroy()`. C++ pseudocode detailing the creation, usage, and destruction of streams is shown in Listing 1.4.

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

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

// Launch MyKernelA using the 0th stream on a grid of one one-thread block with 0 bytes of dynamically allocated shared memory
MyKernelA<<< 1, 1, 0, stream[ 0 ] >>>( inputVariableA )

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

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

Listing 1.4: C++ pseudocode showcasing the creation, usage, and destruction of two streams. Taken from *CUDA C++ Programming Guide* [3].

Assuming that the total local memory required by both kernels called in Listing 1.4 can be provided by a device at a point in time, they will be executed in parallel. For a visualization of concurrent kernel execution see Figure 1.9.

1.2.5 Parallel Reduction

This section aims to present a parallel computation concept relevant to the project. As stated earlier in Section 1.1.1, the parallelization of some tasks is not straightforward. An example of such a task is finding the maximum value in an array.

The naive procedure for finding the maximum value is to iterate through the array, compare each encountered element to a temporary variable, and save the larger of the two compared values into the variable. However, this procedure is strictly sequential and thus inefficient when performed on the device in its current form. In this case, *parallel reduction* can be used to parallelize the problem.

Parallel reduction is a means of reducing the values of an array into a single value using an associative binary operator [25]. The operators often used in parallel reduction are, for example, min, max, arg min, arg max, sum, etc. To clearly explain parallel reduction, the max operator will be assumed.

In the context of CUDA, finding the maximum value of an array stored in global memory is a two-step procedure:

1. A kernel is launched in which each thread block copies a subarray into shared memory and finds its maximum value. The block then saves the maximum value into another array stored in global memory. This process is repeated for the array comprising of per-block maximum values until the number of elements required for comparison is smaller than the maximum number of threads a block can have.

2. The kernel is launched again on a grid comprising a single thread block. This allows loading the entire array into the block's shared memory which means that the maximum value produced will be the largest in the initial array.

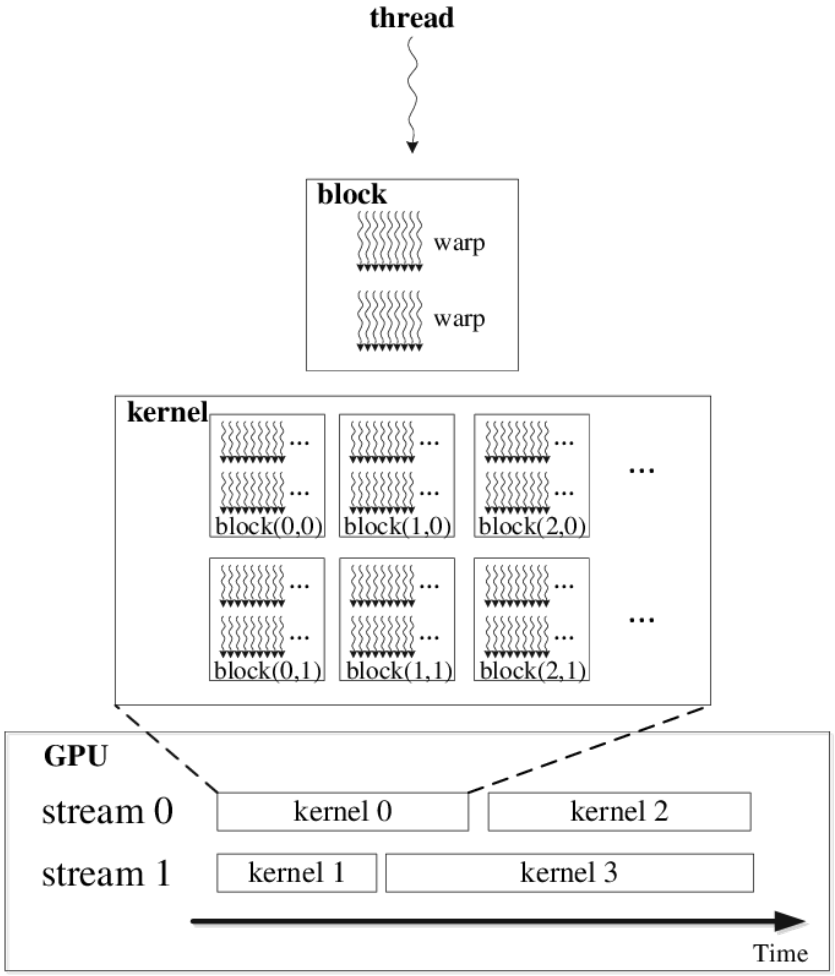

Figure 1.9: Visualization of concurrent kernel execution. Taken from *POM.gpu-v1.0* [24].

To clearly explain parallel reduction, an example detailing the kernel will be described, i.e., finding the maximum value of an array stored in shared memory.

In short, parallel reduction performs iterations of simultaneous pair-comparisons as shown in Figure 1.10.

As can be seen in Figure 1.10, in every iteration, each thread compares two values and then saves the larger of the two into the array. Starting with the 2nd iteration, to reduce the workload, the number of values used is halved in every iteration. Specifically, only the values saved by threads in the previous iteration are evaluated. Thus, starting with the 2nd iteration, the number of threads used in each iteration is halved as they are no longer needed.

Note that the memory-accessing procedure of parallel reduction presented in Figure 1.10 is referred to as *interleaved addressing*. The reason for this is that the values are stored in shared memory which makes this usage of threads cause shared memory bank conflicts. Simply put, a shared memory bank conflict signifies that threads were not used optimally when accessing shared memory - for a detailed description of shared memory bank conflicts see *Parallel LU Decomposition for the GPU* [2]. Parallel reduction with an alternative memory-accessing procedure known as *sequential addressing* is shown in Figure 1.11.

For parallel reduction in CUDA, sequential addressing has been shown to be 2x faster on average than interleaved addressing [26] - for more details see the performance comparison presented in

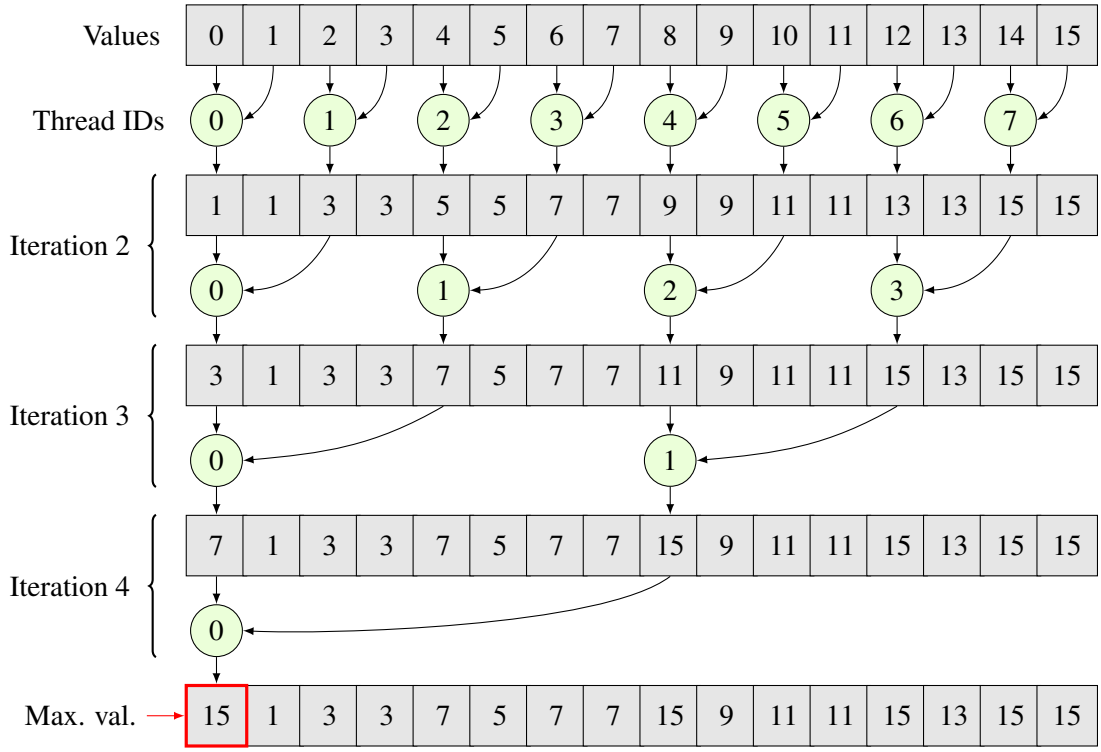

Figure 1.10: Visualization of finding the maximum value in a 16-element array stored in shared memory using parallel reduction. The gray squares contain values of the array. Each row of gray squares shows the array contents in a given iteration. The light green circles represent threads; the number inside each circle is the thread ID. The maximum value found is in the red-bordered square. This figure was created based on the example in *Optimizing Parallel Reduction in CUDA* by Harris, M. [26].

Optimizing Parallel Reduction in CUDA [26].

1.3 Iterative Crout's Method With Partial Pivoting (ICMPP)

The core aspect of HPC is solving advanced computation problems across various fields. The specific types of problems HPC is used to solve are, for example, developing new drugs, deciphering the functioning of the human brain, developing driverless cars, etc. [27]. Solving complex tasks can involve using a wide range of programs that often rely on dependencies to perform fundamental tasks, e.g., solving a system of linear equations, efficiently.

The roots of linear equation solving can be traced back to ancient Chinese mathematics books from around 100 BC [28]. Since then, it has become a fundamental component of numerical linear algebra. Owing to its early discovery and wide range of uses, many different methods have been developed - Gaussian elimination being arguably the most well-known. However, this project focuses on the use of Lower-Upper decomposition with partial pivoting (LUP) and substitution to solve linear equations. Specifically, the LUP method selected was the iterative variant of Prescott Durand Crout's matrix decomposition method, referred to - in the context of this project - as the *Iterative Crout Method* (ICM). Note that in the context of this project, ICM does not include pivoting; a modification of ICM that includes pivoting will be referred to as *Iterative Crout's Method with partial pivoting* (ICMPP).

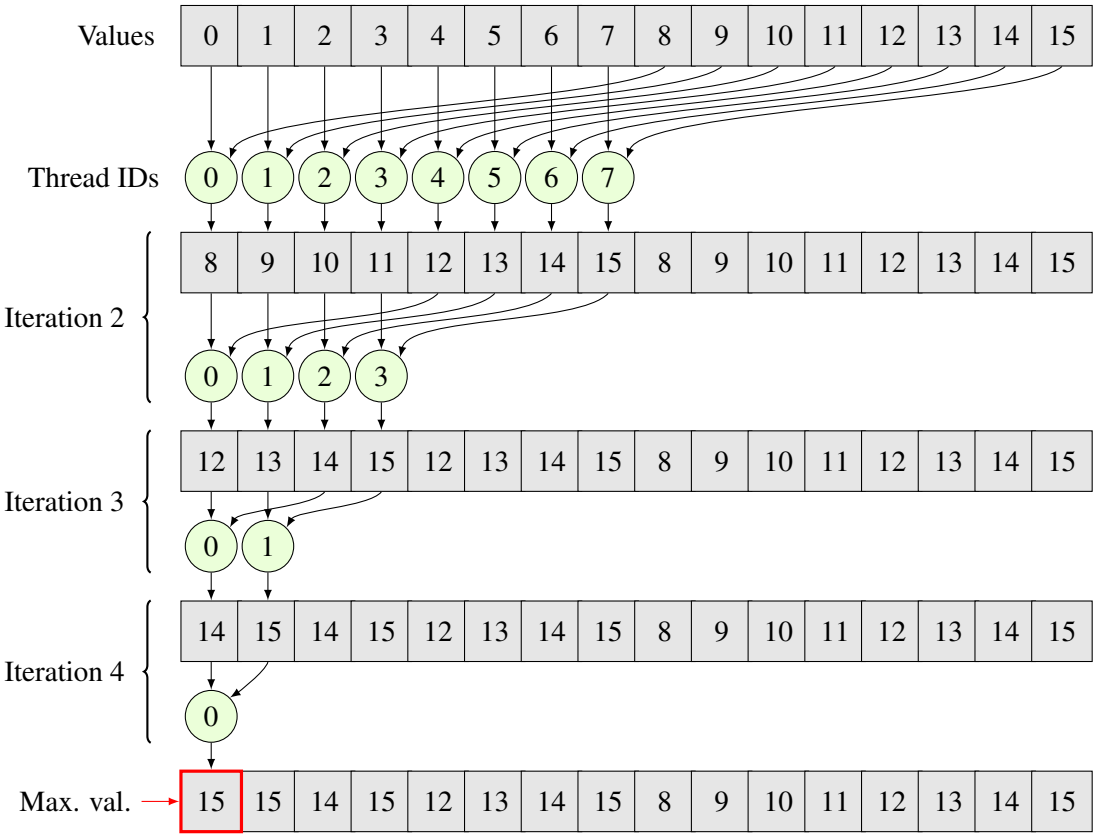

Figure 1.11: Visualization of finding the maximum value in a 16-element array stored in shared memory using parallel reduction with *sequential addressing*. The gray squares contain values of the array. Each row of gray squares shows the array contents in a given iteration. The light green circles represent threads; the number inside each circle is the thread ID. The maximum value found is in the red-bordered square. This figure was created based on the example in *Optimizing Parallel Reduction in CUDA* by Harris, M. [26].

First, LUP and its use when solving a system of linear equations will be briefly described. Then, Crout's Method with partial pivoting (CMPP) will be introduced and, finally, ICMPP will be described.

1.3.1 LU Decomposition With Partial Pivoting (LUP)

It can be argued that LUP is simply Lower-Upper decomposition (LU) with the added feature of partial pivoting. To clearly explain LUP, LU will first be introduced.
LU is a procedure that factors an input matrix into the product of a lower-triangular matrix and an upper-triangular matrix

$$\mathbf{A} = \mathbf{LU}\,, \tag{1.1}$$

where $\mathbf{A}, \mathbf{L}, \mathbf{U} \in \mathbb{R}^{n\times n}$ ($n \in \mathbb{N}$). Specifically, the lower-triangular matrix $\mathbf{L}$ has the form

$$\mathbf{L} = \begin{bmatrix} l_{1,1} & 0 & \dots & \dots & 0 \\ l_{2,1} & l_{2,2} & \ddots & & \vdots \\ l_{3,1} & l_{3,2} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & l_{n,n} \end{bmatrix}, \tag{1.2}$$

and the upper-triangular matrix $\mathbf{U}$ has the form

$$\mathbf{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ 0 & u_{2,2} & u_{2,3} & \dots & u_{2,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & u_{n-1,n} \\ 0 & \dots & \dots & 0 & u_{n,n} \end{bmatrix}. \tag{1.3}$$

However, the above-introduced version of LU is susceptible to failure if $\mathbf{A}$ is not strongly regular. This limitation can be overcome by properly ordering the rows and columns of $\mathbf{A}$ - referred to as *LU decomposition with full pivoting*. Furthermore, the factorization is numerically stable in practice even if only rows are permuted [29] - referred to as *LU decomposition with partial pivoting* (LUP).

The decomposition performed by LUP is identical to that of LU with the exception of a permutation matrix being added to keep track of row permutations. In matrix form, LUP is written as

$$\mathbf{PA} = \mathbf{LU}\,, \tag{1.4}$$

where $\mathbf{P} \in \{0,1\}^{n\times n}$ ($n \in \mathbb{N}$) is a permutation matrix, i.e., each row and column of $\mathbf{P}$ has only one entry equal to 1 and all other entries are equal to 0. Alternatively, LUP can also be written as

$$\mathbf{A} = \mathbf{LUP}\,, \tag{1.5}$$

according to *Necessary And Sufficient Conditions For Existence of the LU Factorization of an Arbitrary Matrix* [30].

Next, the usage of LUP when solving a system of linear equations will be described. A system of $n \in \mathbb{N}$ linear equations and n unknowns can be written in matrix form as

$$\mathbf{Ax} = \mathbf{b}\,, \tag{1.6}$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a coefficient matrix, $\mathbf{x} \in \mathbb{R}^n$ is a vector of unknowns, and $\mathbf{b} \in \mathbb{R}^n$ is a vector containing right-hand side values.

Note that a system of $n \in \mathbb{N}$ linear equations and n unknowns with $m \in \mathbb{N}$ right-hand sides can be written in matrix form as

$$\mathbf{AX} = \mathbf{B}\,, \tag{1.7}$$

where $\mathbf{X} \in \mathbb{R}^{n \times m}$ is a matrix of unknowns and $\mathbf{B} \in \mathbb{R}^{n \times m}$ a matrix of right-hand sides.

Solving a system of linear equations using LUP is a two-step process [2]:

1. *Decomposition* - assuming a system of $n \in \mathbb{N}$ linear equations and n unknowns defined in Equation 1.6, matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ permuted by a permutation matrix $\mathbf{P} \in \{0, 1\}^{n \times n}$ is decomposed into the product of a lower-triangular matrix $\mathbf{L} \in \mathbb{R}^{n \times n}$ and an upper-triangular matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$ as shown in Equation 1.4.
   To use **LU**, both sides of Equation 1.6 must first be left-multiplied by **P**

   $$\mathbf{PAx} = \mathbf{Pb}\,. \tag{1.8}$$

   Then, substituting **LU** for **PA** in Equation 1.8 yields

   $$\mathbf{LUx} = \mathbf{Pb}\,. \tag{1.9}$$

2. *Substitution* - the system presented in Equation 1.9 is then solved in two steps:

   (a) Forward substitution - solve $\mathbf{Ly} = \mathbf{Pb}$ where only vector $\mathbf{y} \in \mathbb{R}^n$ is not known. Note that, in practice, vector **b** is permuted before this system is solved.

   (b) Backward substitution - solve $\mathbf{Ux} = \mathbf{y}$ where only vector $\mathbf{x} \in \mathbb{R}^n$ is not known.

Once **x** is populated with the solution, it can be directly applied to the system presented in Equation 1.6, i.e., there is no need for additional permuting.

Note that **b** is only required in Step 2, i.e., it is not used to decompose **A**. As mentioned in *Parallel LU Decomposition for the GPU* [2] and in *Linear Equations and Eigensystems* [31], this presents an advantage for LUP over Gaussian elimination if multiple right-hand sides are present. The reason behind this is that with Gaussian elimination, only one right-hand side can and must be present for the decomposition. Thus, to solve a system with multiple right-hand sides, Gaussian elimination must decompose **A** for each side. On the other hand, LUP requires **A** to be decomposed only once.

While there are many different approaches to LUP, this project focuses on *Crout's Method with partial pivoting* (CMPP) and *Iterative Crout's Method with partial pivoting* (ICMPP).

1.3.2 Crout's Method With Partial Pivoting (CMPP)

This section aims to introduce an LUP algorithm known as *Crout's Method with partial pivoting* (CMPP). CMPP's base form, Crout's Method (CM), is also referred to as *Crout's matrix decomposition* or *Crout's factorization* and was developed by Prescott Durand Crout [32] in the 20th century.

In the context of this project, CM refers to the method without partial pivoting and whereas CMPP refers to the method with partial pivoting.

When it comes to LUP algorithms, a distinctive feature of CM is that it generates a unit upper-triangular matrix $\mathbf{U} \in \mathbb{R}^{n \times n}$ ($n \in \mathbb{N}$), i.e., one where all the elements on its main diagonal are equal to 1:

$$\mathbf{U} = \begin{bmatrix} 1 & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ 0 & 1 & u_{2,3} & \dots & u_{2,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & u_{n-1,n} \\ 0 & \dots & \dots & 0 & 1 \end{bmatrix}. \tag{1.10}$$

Algorithm The core part of the algorithm for decomposing matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ ($n \in \mathbb{N}$) into **LUP** - where $\mathbf{L}, \mathbf{U} \in \mathbb{R}^{n \times n}$ ($n \in \mathbb{N}$) and $\mathbf{P} \in \{0, 1\}^{n \times n}$ - consists of formulas for computing the elements of $\mathbf{L}$ and $\mathbf{U}$ [32]:

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

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

$$u_{i,j} = 1 \qquad i = j\,,$$

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

The algorithm itself consists of the following steps (assuming input matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ where $n \in \mathbb{N}$) [32, 33]:

1 Set j equal to 1.

2 If j is greater than n, then stop the execution as matrices $\mathbf{L}$ and $\mathbf{U}$ have been successfully computed.

3 Compute values from $l_{j,j}$ to $l_{n,j}$ in column j of $\mathbf{L}$.

4 Pivot row j:

4.1 In the values computed in step **3**, find the index (p) of the element largest in absolute value:

$$p = \arg\max_{k} \left\{ \left| l_{k,j} \right| : k = j, \dots, n \right\}. \tag{1.13}$$

4.2 If j is not equal to p, swap rows j and p in matrices $\mathbf{L}$, $\mathbf{U}$, and $\mathbf{P}$.

4.3 If $l_{j,j}$ is equal to 0, then the decomposition algorithm has failed as the matrix is singular.

5 Compute values from $u_{j,j+1}$ to $u_{j,n}$ in row j of $\mathbf{U}$.

6 Increment j by 1 and go to step **2**.

See Figure 1.12 for a visualization of the algorithm's advance and Listing 1.5 for a pseudocode implementing the algorithm.

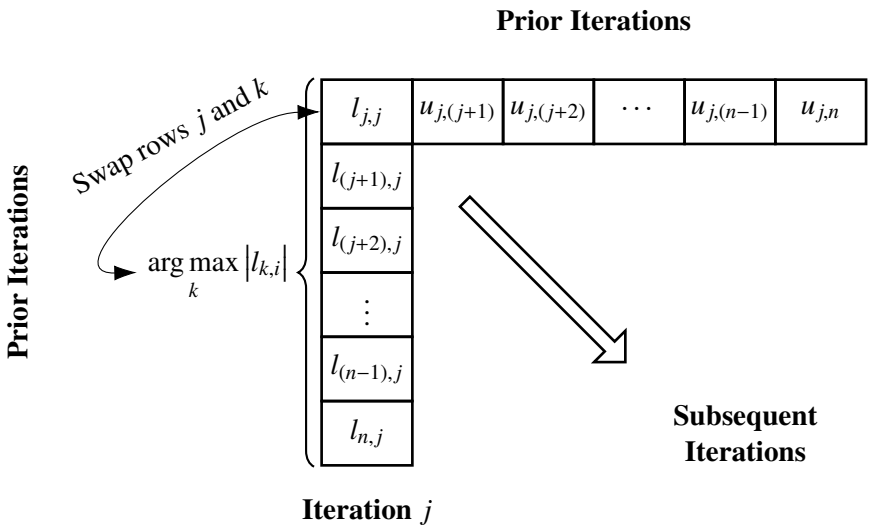

Figure 1.12: Visualization of the advance of CMPP's algorithm. First, elements $l_{j,j}$ to $l_{n,j}$ in column j of $\mathbb{L}$ are computed. Then, row j is pivoted. Its row is swapped with the row containing the largest absolute value of an element in column j, starting from $l_{j,j}$ and ending at $l_{n,j}$, both inclusive. Finally, elements $u_{j,j+1}$ to $u_{j,n}$ in row i of $\mathbb{U}$ are computed. Adapted from *Crout's LU Factorization* by Vismor [34].

```
1  void swapRows( M, row1, row2, n )
2  {
3    for( Index j = 0; j < n; ++j ) {
4      double temp = M[ row1 ][ j ];
5      M[ row1 ][ j ] = M[ row2 ][ j ];
6      M[ row2 ][ j ] = temp;
7    }
8  }
9
10 int pivotRowOfMatrix( j, M, piv, n )
11 {
12   // Find row below the j-th row with max. value in the j-th column
13   double maxAbs = abs( M[ j ][ j ] );
14   int pivRow = j;
15
16   for( Index i = j + 1; i < n; ++i ) {
17     double absElem = abs( M[ j ][ j ] );
18     if( absElem > maxAbs ) {
19       maxAbs = absElem;
20       pivRow = i;
21     }
22   }
23
24   if( pivRow != j ) {  // swap rows j and pivRow
25     swapRows( M, j, pivRow, n );
26     piv( j ) = pivRow + 1;
27   }
28
29   return pivRow;
30 }
31
32 void cmpp( A, L, U, piv, n )
33 {
34   int i, j, k;
```

```
35    double sum = 0;
36
37    // Fill main diagonal of U with 1s
38    for( i = 0; i < n; ++i ) {
39      U[ i ][ i ] = 1;
40    }
41
42    // Loop through the main diagonal
43    for( j = 0; j < n; ++j ) {
44
45      // Compute column j in L
46      for( i = j; i < n; ++i ) {
47        sum = 0;
48        for( k = 0; k < j; ++k ) {
49          sum += L[ i ][ k ] * U[ k ][ j ];
50        }
51
52        L[ i ][ j ] = A[ i ][ j ] - sum;
53      }
54
55      // Pivot row j
56      int pivRow = pivotRowOfMatrix( j, L, piv, n );
57      // Swap rows of remaining matrices to maintain the same ordering
58      if( pivRow != j ) {
59        swapRows( A, j, pivRow, n );
60        swapRows( U, j, pivRow, n );
61      }
62
63      // Decomposition failed as division by zero would occur on line 76
64      if( L[ j ][ j ] == 0 ) {
65        printf( "-!> Cannot decompose singular Matrix A!" );
66        exit( EXIT_FAILURE );
67      }
68
69      // Compute row j in U
70      for( i = j; i < n; ++i ) {
71        sum = 0;
72        for( k = 0; k < j; ++k ) {
73          sum = sum + L[ j ][ k ] * U[ k ][ i ];
74        }
75
76        U[ j ][ i ] = ( A[ j ][ i ] - sum ) / L[ j ][ j ];
77      }
78    }
79  }
```

Listing 1.5: C++ pseudocode for CMPP's algorithm that decomposes with partial pivoting an `n`-by-`n` matrix `A`. The two-dimensional arrays `A[n][n]`, `L[n][n]`, and `U[n][n]` represent matrices **A**, **L**, and **U**, respectively. Instead of using a two-dimensional array to represent matrix $\mathbb{P}$ a one-dimensional array `piv` is used. It stores the index of the row that each row was pivoted with, e.g., `piv[0] = 8` signifies that row `0` was swapped with row `8`. Note that, on input, `L` and `U` are assumed to be populated with 0s. Derived from *Algorithm 16* [33] and *Numerical recipes* [32].

From Formulas 1.11 and 1.12, and the pseudocode presented in Listing 1.5, it can be seen that the main part of computing an element in either **L** or **U** (where $\mathbf{L}, \mathbf{U} \in \mathbb{R}^{n \times n}$ ($n \in \mathbb{N}$)) is the sum. For each element, the sum is computed using some elements above and to the left of it. Specifically, elements starting from 1 to min(i, j) (excluding the latter) in row i and column j are multiplied and summed. This dependency of elements is visualized in Figure 1.13.

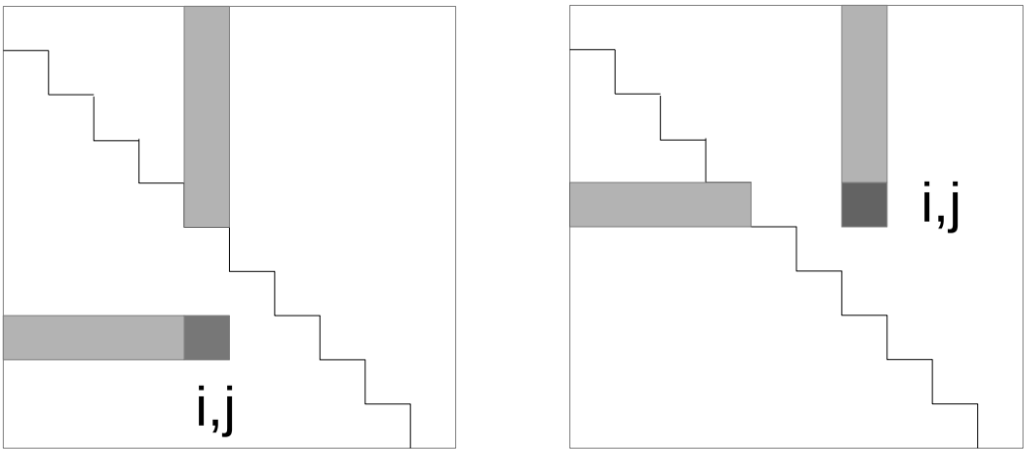

Figure 1.13: Two examples of elements used (light gray square) to compute the sum in the formula of elements $l_{i,j}$ and $u_{i,j}$. The dark gray squares represent the elements being computed. The lower triangle of the matrix consists of the corresponding elements of the lower triangle from **L** while the upper triangle consists of the corresponding elements of the upper triangle from **U**. To compute the sum, only elements with indices in the interval $[1, \min(i, j))$, i.e., excluding the right boundary, are used in the element's row and column. Taken from *Parallel LU Decomposition for the GPU* [2] and *Fine-Grained Parallel Incomplete LU Factorization* [35].

In summary, an element in row i and column j is dependent on elements with indices $[1, \min(i, j))$ from its row and elements with indices $[1, \min(i, j))$ from its column. This means that CMPP's algorithm is inherently sequential, which seemingly limits its potential for parallelization.

Thus far, the variation of CMPP discussed will either produce an exact solution in a finite amount of steps or, if the matrix is singular, it will fail. In terms of numerical methods such a method is referred to as *direct*.

An alternative group of methods to direct methods is known as iterative methods. Unlike direct methods, iterative methods converge to a solution, but there is no guarantee that the exact solution will be obtained. An example of an iterative variant of CMPP: *Iterative Crout's Method with partial pivoting* (ICMPP) will be described in the next part.

1.3.3 Iterative Crout's Method With Partial Pivoting (ICMPP)

Seeing as the potential for parallelization is seemingly limited for CMPP's algorithm, an alternative approach can be used. One such alternative, put forward by Anzt, H.; Ribizel, T.; Flegar, G.; Chow, E.; Dongarra, J. in *ParILUT - A Parallel Threshold ILU for GPUs* [36], involves using the formulas of CM in an iterative method. Although the method proposed by the authors generates incomplete factorization, i.e., some nonzero elements are omitted during factorization, its principle can be extracted to create an iterative decomposition method that produces a complete factorization. As detailed in *Parallel LU Decomposition for the GPU* [2], the complete-factorization approach converges to a sufficiently approximate solution of $\mathbf{A} = \mathbf{LU}$. This approach, labeled as *Iterative Crout's Method* (ICM), has been modified to include partial pivoting, thus creating *Iterative Crout's Method with partial pivoting* (ICMPP).

The decomposition of matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ into the product of matrices $\mathbf{L}, \mathbf{U} \in \mathbb{R}^{n \times n}$, and a permutation matrix $\mathbf{P} \in \{0, 1\}^{n \times n}$ using ICMPP consists of the following steps [2]:

1 Create an initial estimate of matrices $\mathbf{L}^0$ and $\mathbf{U}^0$ using $\mathbf{A}$:

$$\begin{aligned} l^0_{i,j} &= a_{i,j} & u^0_{i,j} &= 0 & i < j\,, \\ u^0_{i,j} &= a_{i,j} & l^0_{i,j} &= 0 & i > j\,, \\ u^0_{i,j} &= 1 & & & i = j\,, \end{aligned}$$

and initialize $\mathbf{P}$ as the identity matrix.

2 Denote the current iteration as $t \in \mathbb{N}$ and initialize it to 1.

3 Compute $\mathbf{L}^t$ using Formula 1.11:

$$l^t_{i,j} = a_{ij} - \sum_{k=1}^{j-1} l^{t-1}_{i,k} u^{t-1}_{k,j} \qquad i \geq j\,.$$

4 Take the first $j \in \widehat{n}; n \in \mathbb{N}$ that satisfies the condition $\left| l^{t-1}_{j,j} - l^t_{j,j} \right| < \epsilon$ (where ϵ denotes a tolerance, e.g., 0.001) and *iteratively converge* all elements in $\mathbf{L}^t$ below row j from column 1 to column j (including both boundaries), i.e., $l^t_{b,c}$ where $j < b \leq n$ and $1 \leq c \leq j$. If no such j satisfies the condition, proceed to Step **6**. In this context, *iteratively converge* signifies to continue computing the mentioned elements using Formula 1.11 until they all satisfy the condition $\left| l^{t-1}_{b,c} - l^t_{b,c} \right| < \epsilon$.

5 Pivot row j (same as Step **4** in the CMPP algorithm description on page 32):

5.1 Find the index (p) of the element largest in absolute value under element $l^t_{j,j}$:

$$p = \arg\max_k \left\{ \left| l^t_{k,j} \right| : k = j, \ldots, n \right\}.$$

5.2 If j is not equal to p, swap rows j and p in matrices $\mathbf{L}^t$, $\mathbf{U}^t$, and $\mathbf{P}$.

5.3 If $l^t_{j,j}$ is equal to 0, then the decomposition procedure has failed as the matrix is singular.

6 Compute $\mathbf{U}^t$ using the values of $\mathbf{L}^t$ computed in earlier steps according to the formula provided in Equation 1.12:

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

7 If the absolute difference between $\mathbf{L}^{t-1}$ and $\mathbf{L}^t$, or between $\mathbf{U}^{t-1}$ and $\mathbf{U}^t$, exceeds tolerance ϵ, then increment t by 1 and return to Step **3**.

8 The algorithm has converged to an approximate solution of $\mathbf{A} = \mathbf{LUP}$.

While the instructions in Steps **3** to **7** must be performed consecutively, the operations in each step can be executed in parallel:

- Step **3** - the element $l^t_{i,j}\,(i, j \in \widehat{n}; n \in \mathbb{N})$ is dependent only on elements from matrices $\mathbf{A}$, $\mathbf{L}^{t-1}$, and $\mathbf{U}^{t-1}$. Since the values in the first matrix are constant - excluding the swapping of rows - and the last two matrices were set in the previous iteration, no elements from $\mathbf{L}^t$ are dependent on each other. Thus, all elements $l^t_{i,j}$ can be computed simultaneously.

- Step **4** - the absolute-value condition can be checked in parallel, and, similarly to Step **3**, the iterative convergence is computed according to the same formula.
- Step **5** - the max operation mentioned in Section 1.2.5 can be adapted to suit the needs of arg max used in Step **5.1**. While the swapping of elements in two rows can be performed in parallel, Steps **5.2** and **5.3** must be executed consecutively.
- Step **6** - the element $u^t_{i,j}\,(i, j \in \widehat{n}; n \in \mathbb{N})$ is dependent only on elements from matrices $\mathbf{A}$, $\mathbf{L}^t$, and $\mathbf{U}^{t-1}$. Matrices $\mathbf{A}$ and $\mathbf{U}^{t-1}$ were covered in the description of parallelization of Step **3** and matrix $\mathbf{L}^t$ was finalized in Steps **3** to **5**. Thus, since no elements from $\mathbf{U}^t$ are dependent on each other, they can be computed simultaneously.
- Step **7** - since the evaluation of the convergence rule has no dependencies, it is entirely parallelizable.

Note that due to the nature of iterative methods, it is not possible to accurately predict the number of iterations needed to converge to an approximate solution. Thus, the performance of ICMPP may heavily depend on the nonzero element structure of matrix $\mathbf{A}$.

To summarize, in Sections 1.1 and 1.2 a high-performance parallel computing system was introduced in the form of Nvidia GPUs manageable by CUDA. Then, at the end of Section 1.3, a parallelizable algorithm was presented. The combination of the aforementioned parts will be the main focus of the following chapters.

Conclusion

TODO

Bibliography

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

[2] ČEJKA, L. *Parallel LU Decomposition for the GPU*. Prague, 2022. Research Assignment. Czech Technical University in Prague.

[3] NVIDIA, C. *CUDA C++ Programming Guide: Design Guide* [online]. Nvidia, 2023 [visited on 2023-04-30]. Available from: https://docs.nvidia.com/cuda/archive/12.0.0/pdf/CUDA_C_Programming_Guide.pdf.

[4] NVIDIA, C. *CUDA C++ Best Practices Guide: Design Guide* [online]. Nvidia, 2022 [visited on 2023-04-30]. Available from: https://docs.nvidia.com/cuda/archive/12.0.0/pdf/CUDA_C_Best_Practices_Guide.pdf.

[5] SANGLARD, F. *A HISTORY OF NVIDIA STREAM MULTIPROCESSOR* [online]. [visited on 2023-04-30]. Available from: https://fabiensanglard.net/cuda/.

[6] GIGABYTE. *TDP: What is it?* [online]. GIGABYTE, 2023 [visited on 2023-04-30]. Available from: https://www.gigabyte.com/Glossary/tdp.

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

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

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

[10] NVIDIA, C. *NVIDIA H100 Tensor Core GPU Architecture: EXCEPTIONAL PERFORMANCE, SCALABILITY, AND SECURITY FOR THE DATA CENTER* [online]. 2022. [visited on 2023-04-30]. Available from: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

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

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

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

[14] *Introduction to GPGPU and CUDA Programming: SIMT and Warp* [online]. Cornell University, 2023 [visited on 2023-05-01]. Available from: https://cvw.cac.cornell.edu/gpu/simt_warp.

[15] GROTE, P. *Lock-based Data Structures on GPUs with Independent Thread Scheduling*. Berlin, 2020. Bachelor Thesis. Technischen Universität Berlin.

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

[17] GROVER, V.; LIN, Y. *Using CUDA Warp-Level Primitives* [online]. Nvidia, 2023 [visited on 2023-05-01]. Available from: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[18] THOMAS-COLLIGNON, G.; MICIKEVICIUS, P. *VOLTA Architecture and performance optimization* [online]. Nvidia [visited on 2023-05-01]. Available from: https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf.

[19] GATES, M. *CUDA syntax* [online]. The University of Tennessee, Knoxville, Tennessee 37996: The University of Tennessee, 2023 [visited on 2023-05-03]. Available from: https://icl.utk.edu/~mgates3/docs/cuda.html.

[20] GUPTA, P. *CUDA Refresher: The CUDA Programming Model* [online]. Nvidia, 2023 [visited on 2023-05-03]. Available from: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model.

[21] HARRIS, M. *Using Shared Memory in CUDA C/C++* [online]. Nvidia, 2023 [visited on 2023-05-04]. Available from: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc.

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

[23] HERNÁNDEZ, M.; GUERRERO, G. D.; CECILIA, J. M.; GARCÍA, J. M.; INUGGI, A.; JBABDI, S.; BEHRENS, T. E. J.; SOTIROPOULOS, S. N. Correction: Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs. *PLOS ONE* [online]. 2015, vol. 10, no. 6 [visited on 2023-05-04]. ISSN 1932-6203. Available from DOI: 10.1371/journal.pone.0130915.

[24] XU, S.; HUANG, X.; OEY, L.-Y.; XU, F.; FU, H.; ZHANG, Y.; YANG, G. POM.gpu-v1.0: a GPU-based Princeton Ocean Model. *Geoscientific Model Development* [online]. 2015, vol. 8, no. 9, pp. 2815–2827 [visited on 2023-05-04]. ISSN 1991-9603. Available from DOI: 10.5194/gmd-8-2815-2015.

[25] KIRK, D. B.; HWU, W.-m. W. Chapter 6 - Performance Considerations. In: *Programming Massively Parallel Processors (Second Edition)*. Second. Boston: Morgan Kaufmann, 2013, pp. 123–149. ISBN 978-0-12-415992-1.

[26] HARRIS, M. *Optimizing Parallel Reduction in CUDA* [online]. Nvidia, 2023 [visited on 2023-05-08]. Available from: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[27] *High Performance Computing - best use examples* [online]. 2023. [visited on 2023-05-10]. Available from: https://digital-strategy.ec.europa.eu/en/library/high-performance-computing-best-use-examples.

[28] HART, R. *The Chinese roots of linear algebra*. 1st edition. Baltimore, MD: Johns Hopkins University Press, 2011. ISBN 9780801897559.

[29] TREFETHEN, L. N.; BAU, D. *Numerical linear algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 1997. ISBN 978-0-89871-361-9.

[30] OKUNEV, P.; JOHNSON, C. R. Necessary And Sufficient Conditions For Existence of the LU Factorization of an Arbitrary Matrix. 1997. Available from DOI: 10.48550/arXiv.math/0506382.

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

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

[33] FORSYTHE, G. E. Algorithm 16: crout with pivoting. *Communications of the ACM* [online]. 1960, vol. 3, no. 9, pp. 507–508 [visited on 2023-05-12]. ISSN 0001-0782. Available from DOI: 10.1145/367390.367406.

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

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

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

[37] JÚNIOR, E. *Implementing parallel reduction in CUDA* [online]. 2022. [visited on 2023-05-05]. Available from: https://eximia.co/implementing-parallel-reduction-in-cuda/.

[38] GRAVELL, M. *How I found CUDA, or: Rewriting the Tag Engine–part 2: CUDA: Kernels, Threads, Warps, Blocks and Grids* [online]. 2016. [visited on 2023-05-01]. Available from: https://blog.marcgravell.com/2016/05/how-i-found-cuda-or-rewriting-tag_9.html.

[39] DONGARRA, J.; GATES, M.; HAIDAR, A.; KURZAK, J.; LUSZCZEK, P.; WU, P.; YAMAZAKI, I.; YARKHAN, A.; ABALENKOVS, M.; BAGHERPOUR, N.; HAMMARLING, S.; ŠÍSTEK, J.; STEVENS, D.; ZOUNON, M.; RELTON, S. D. PLASMA. *ACM Transactions on Mathematical Software* [online]. 2019, vol. 45, no. 2, pp. 1–35 [visited on 2022-10-11]. ISSN 0098-3500. Available from DOI: 10.1145/3264491.

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

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

Appendix A

Title

TODO