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
GEM	Gaussian Elimination Method
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 designed to accelerate 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.

```
1 multiplyMatrices(A, B, C):
2   for i from 0 to m - 1:
3     for j from 0 to p - 1:
4       for k from 0 to n - 1:
5         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.

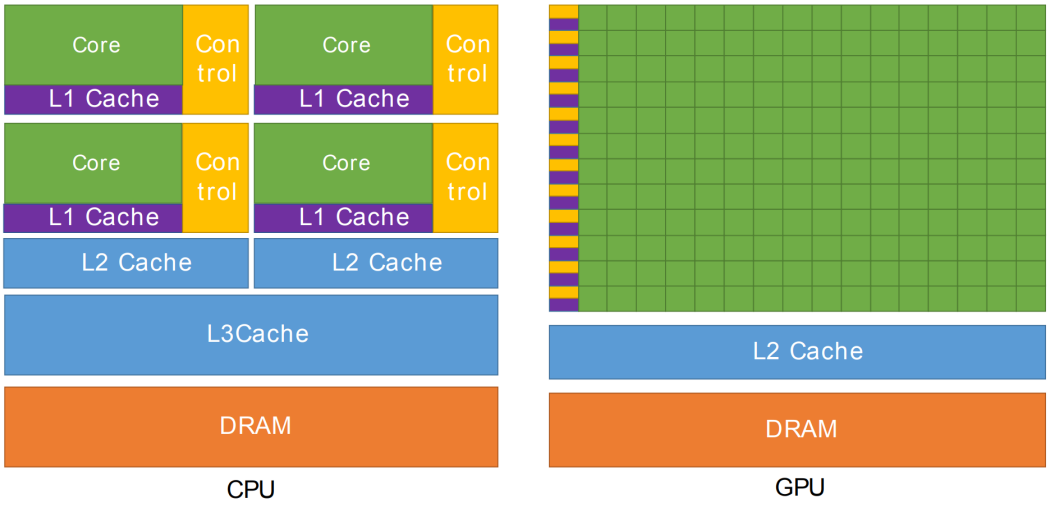

Figure 1.1: Architecture comparison of a CPU and a GPU. Taken from *CUDA C++ Programming Guide* [3].

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,

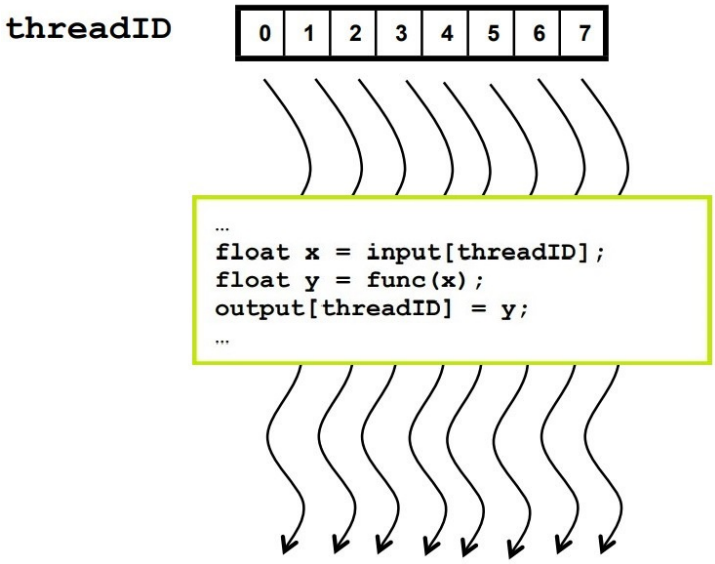

Figure 1.2: Execution of code by an 8-thread warp. The unique ID of each thread is stored in the variable `threadID`. In this example, `threadID` is used to read and write different data. Taken from *Getting Started with CUDA* [13] and *Parallel LU Decomposition for the GPU* [2].

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;
```

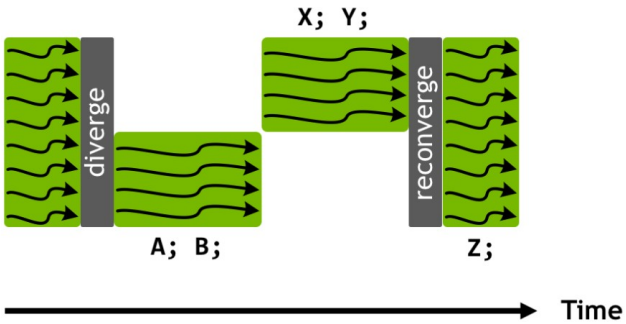

Figure 1.3: Pseudocode showcasing thread divergence in an 8-thread warp. The `threadIdx.x` variable contains each thread's ID in the 1st dimension. In this example, while threads `0` to `3` execute statements `A` and `B`, threads `4` to `7` are idle. Then, threads `4` to `7` execute statements `X` and `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.

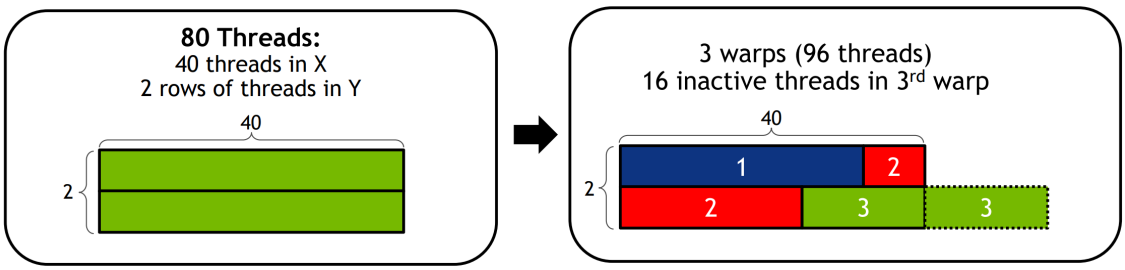

Figure 1.4: Division of threads into warps in a block of 40 by 2 threads. Each green rectangle in the left image represents a row of threads. In the right image, each color represents a unique warp. Taken from *VOLTA Architecture and performance optimization* [18].

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` - The 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` - The 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` - The 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.

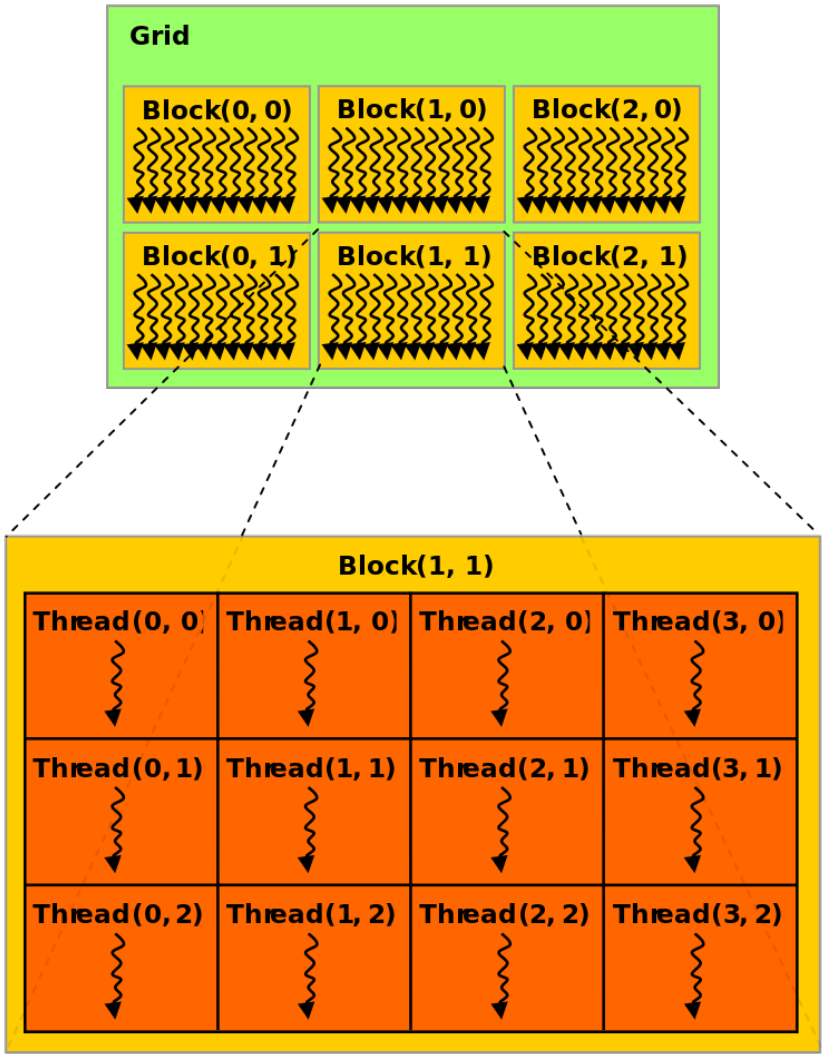

Figure 1.5: Visualization of a 2-by-3 grid comprising 3-by-4 thread blocks. The numbers in the brackets are the IDs of each entity in each dimension. Taken from *CUDA C++ Programming Guide* [3].

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;

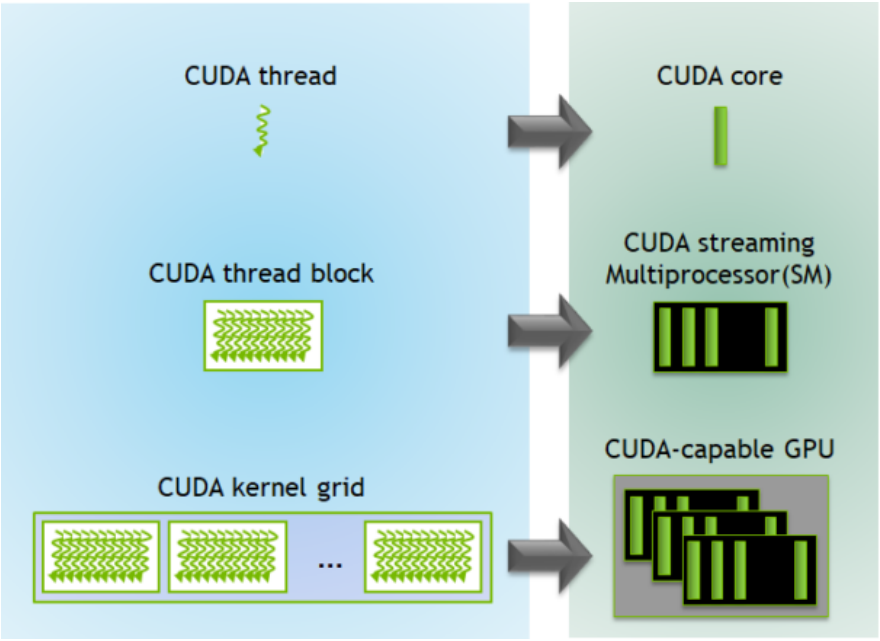

Figure 1.6: Visualization illustrating the execution of CUDA thread structures by different hardware components in an Nvidia GPU. Taken from *CUDA Refresher: The CUDA Programming Model* [20].

- `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.

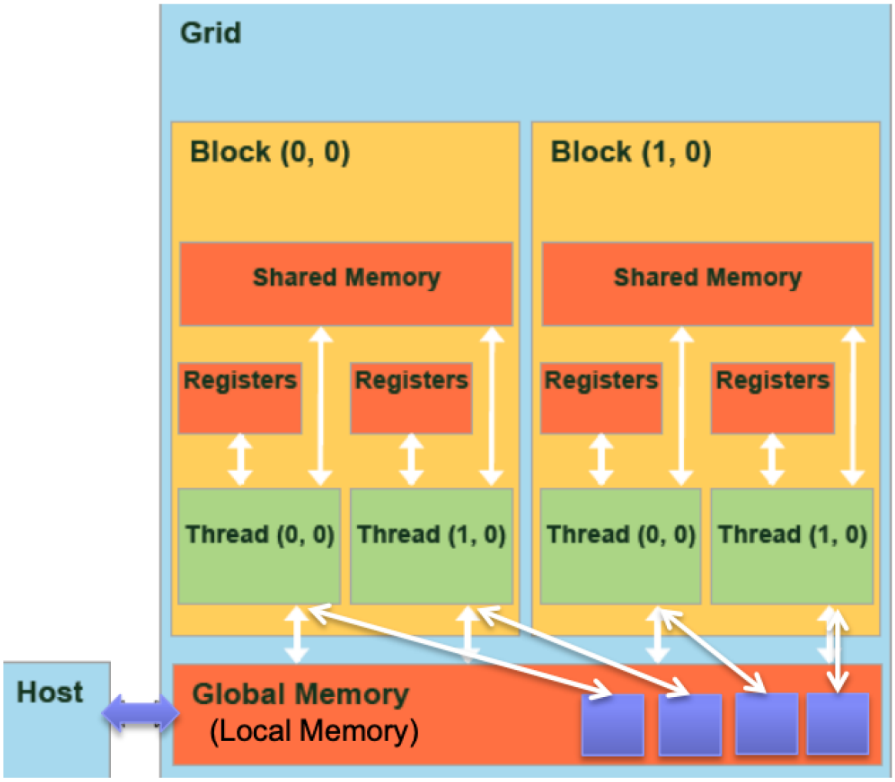

Figure 1.7: CUDA memory structuring with accesses available to each thread structure. Taken from *GPU* by Hsiao, Y. [22].

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

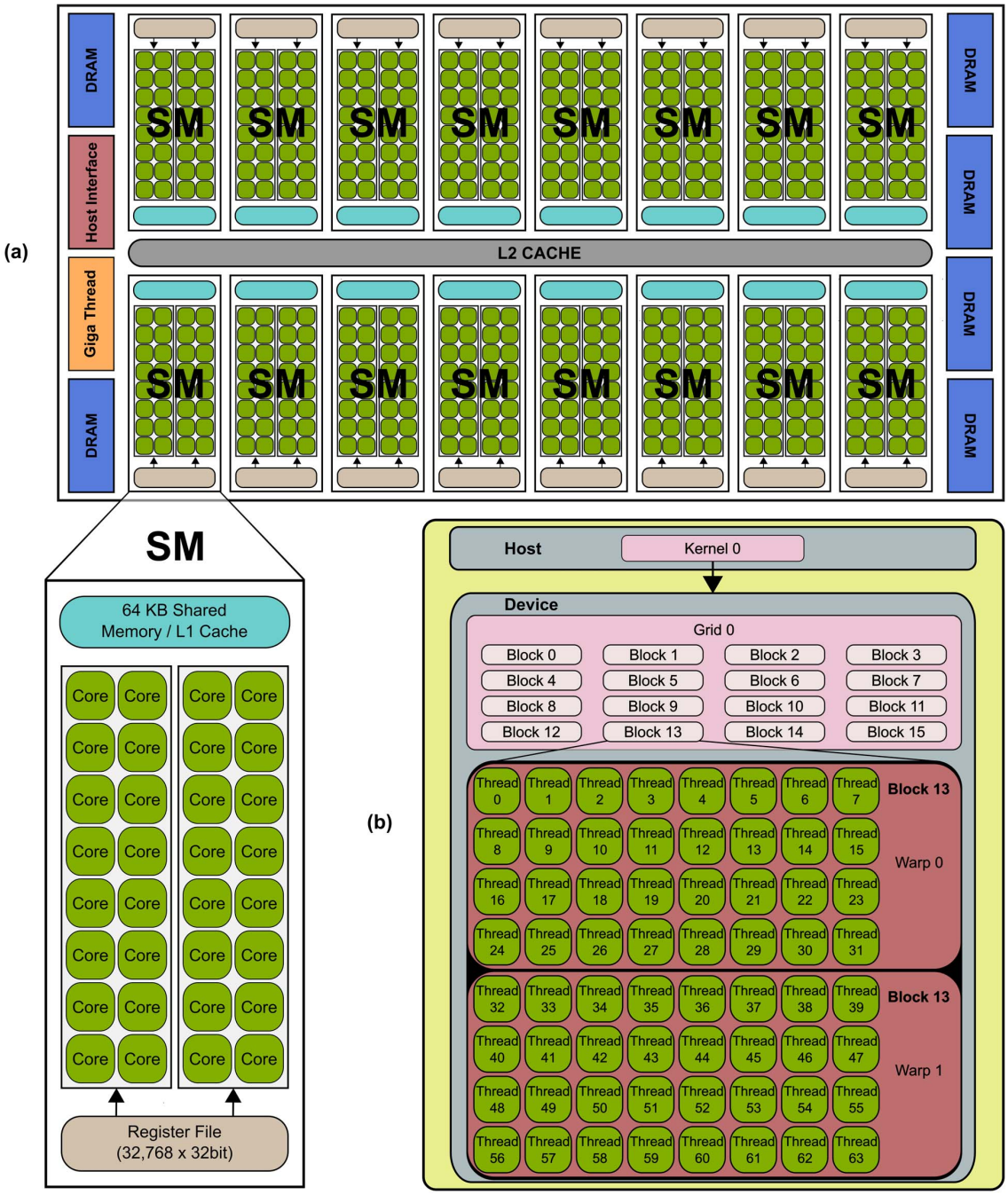

Figure 1.8: Simplified overview of an Nvidia GPU's architecture and the CUDA programming model. Taken from *Correction* [23].

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, 3]:

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. In particular, if kernels are issued from distinct processes, they cannot run concurrently unless CUDA MPS[2] is used [24, 25].

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

[2]CUDA MPS documentation webpage URL: `https://docs.nvidia.com/deploy/mps/index.html`

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 [27]. 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.

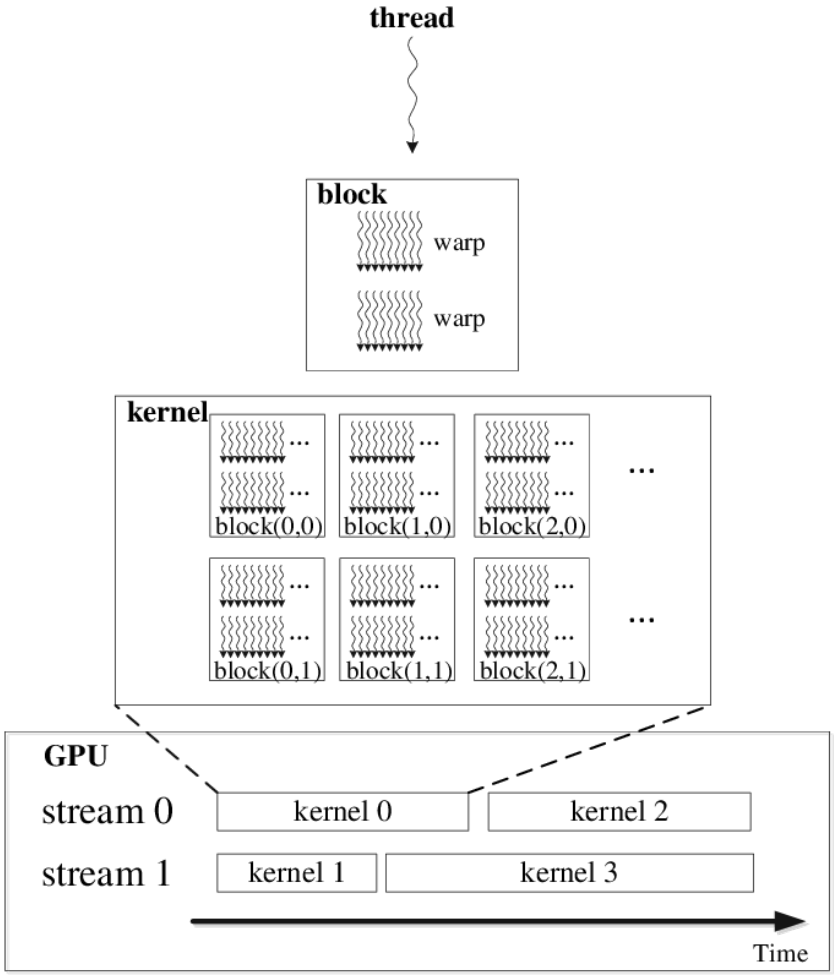

Figure 1.9: Visualization of concurrent kernel execution. Taken from *POM.gpu-v1.0* [26].

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.

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*

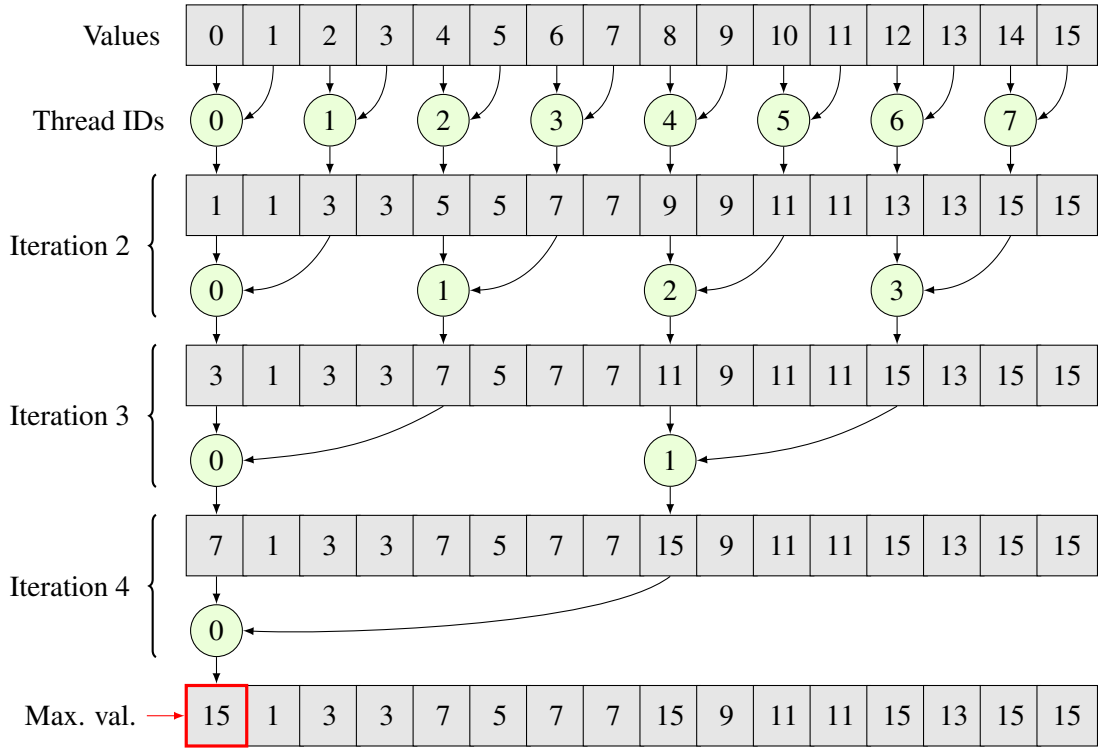

Figure 1.10: Visualization of finding the maximum value in a 16-element array stored in shared memory using parallel reduction. The gray squares contain values of the array. Each row of gray squares shows the array contents in a given iteration. The light green circles represent threads; the number inside 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. [28].

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 [28] - for more details see the performance comparison presented in *Optimizing Parallel Reduction in CUDA* [28].

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. [29]. 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 [30]. 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

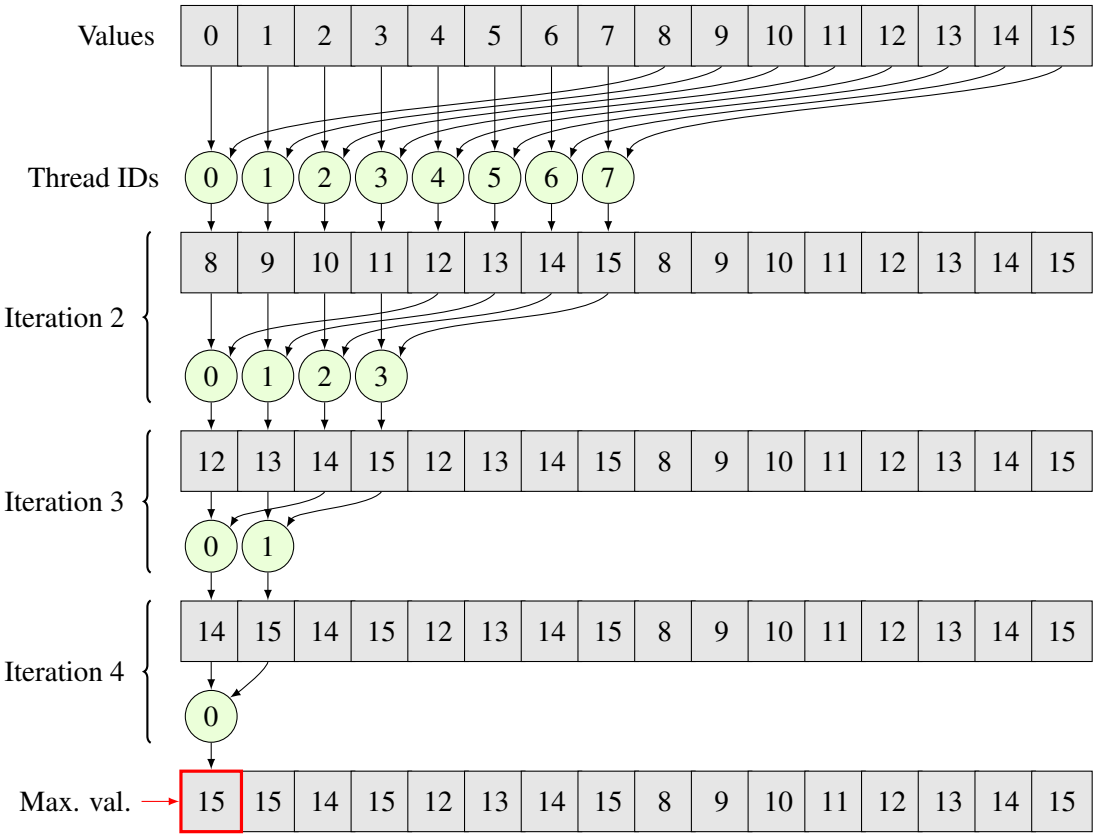

Figure 1.11: Visualization of finding the maximum value in a 16-element array stored in shared memory using parallel reduction with *sequential addressing*. The gray squares contain values of the array. Each row of gray squares shows the array contents in a given iteration. The light green circles represent 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. [28].

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).

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 [31] - 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* [32].

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** contains 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* [33], this presents an advantage for LUP over Gaussian elimination in cases where right-hand sides are not provided at once. For example, when solving time-dependent partial differential equations, each time step presents a system whose right-hand side is determined by the solution of the system belonging to the previous time step. In such cases, Gaussian elimination would need to be performed in its entirety for every new right-hand side, whereas LUP would be used to decompose **A** once, and then the output of the decomposition would be reused for every new right-hand side.

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 [34] 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 $\mathbf{U}$ [34]:

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

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

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

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

The algorithm itself consists of the following steps (assuming input matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ where $n \in \mathbb{N}$) [34, 35]:

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:
   (a) In the values computed in step 3, find the index (p) of the element largest in absolute value:
   $$p = \arg\max_k \left\{\left|l_{k,j}\right| : k = j, \dots, n\right\}. \tag{1.13}$$
   (b) If j is not equal to p, swap rows j and p in matrices $\mathbf{L}$, $\mathbf{U}$, and $\mathbf{P}$.
   (c) 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.

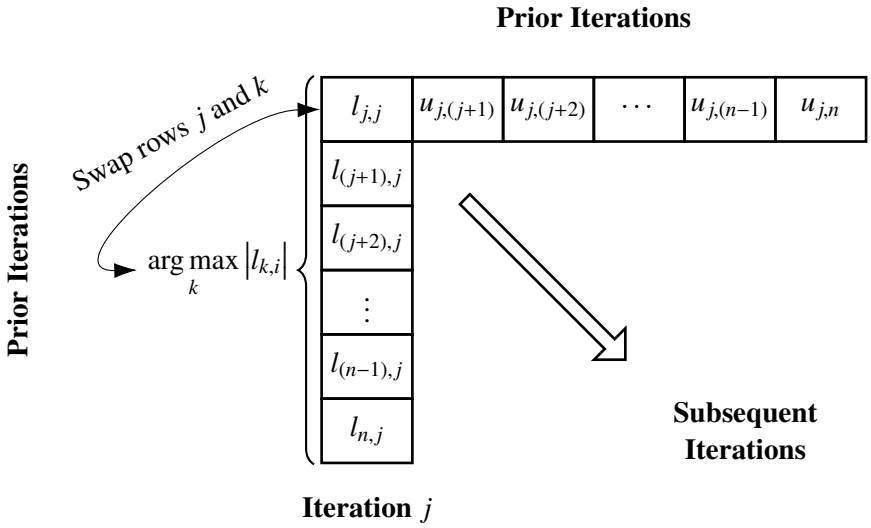

Figure 1.12: Visualization of the advance of CMPP's algorithm. First, elements $l_{j,j}$ to $l_{n,j}$ in column j of $\mathbb{L}$ are computed. Then, row j is pivoted. Its row is swapped with the row containing the largest absolute value of an element in column j, starting from $l_{j,j}$ and ending at $l_{n,j}$, 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 [36].

```
void swapRows( M, row1, row2, n )
{
  for( Index j = 0; j < n; ++j ) {
    double temp = M[ row1 ][ j ];
    M[ row1 ][ j ] = M[ row2 ][ j ];
    M[ row2 ][ j ] = temp;
  }
}

int pivotRowOfMatrix( j, M, piv, n )
{
  // Find row below the j-th row with max. value in the j-th column
  double maxAbs = abs( M[ j ][ j ] );
  int pivRow = j;

  for( Index i = j + 1; i < n; ++i ) {
    double absElem = abs( M[ j ][ j ] );
    if( absElem > maxAbs ) {
      maxAbs = absElem;
      pivRow = i;
    }
  }

  if( pivRow != j ) {  // swap rows j and pivRow
    swapRows( M, j, pivRow, n );
    piv( j ) = pivRow + 1;
  }

  return pivRow;
```

```
}

void cmpp( A, L, U, piv, n )
{
  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;
    }

    // Pivot row j
    int pivRow = pivotRowOfMatrix( j, L, piv, n );
    // Swap rows of remaining matrices to maintain the same ordering
    if( pivRow != j ) {
      swapRows( A, j, pivRow, n );
      swapRows( U, j, pivRow, n );
    }

    // Decomposition failed as division by zero would occur on line 73
    if( L[ j ][ j ] == 0 ) {
      printf( "-!> Cannot decompose singular Matrix A!" );
      exit( EXIT_FAILURE );
    }

    // 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 ];

      U[ j ][ i ] = ( A[ j ][ i ] - sum ) / L[ j ][ j ];
    }
  }
}
```

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* [35] and *Numerical recipes* [34].

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.

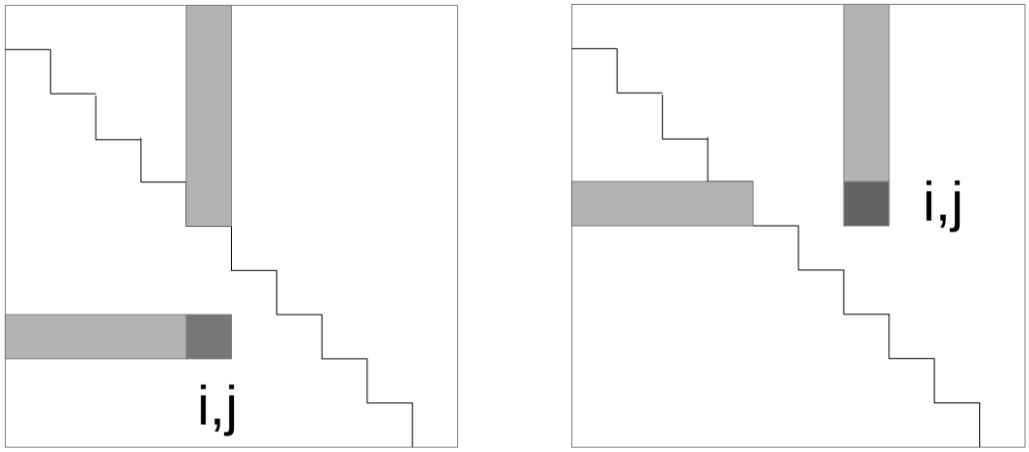

Figure 1.13: Two examples of elements used (light gray square) to compute the sum in the formula of elements $l_{i,j}$ and $u_{i,j}$. The dark gray squares represent the elements being computed. The lower triangle of the matrix consists of the corresponding elements of the lower triangle from **L** while 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* [37].

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* [38], 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):

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

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

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

   (c) 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 maximum 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 5a. While the swapping of elements in two rows can be performed in parallel, Steps 5b and 5c 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.

Chapter 2

Implementation

This chapter presents the implementation of the *Decomposition* project, which serves as a TNL-compatible framework for housing the CMPP and ICMPP algorithms described in Sections 1.3.2 and 1.3.3, respectively. In addition to these algorithms, a set of triangular solvers has been implemented to offer a comprehensive solution for solving systems of equations.

First, the 3rd party libraries used in the project will be presented. Then, the project itself will be detailed. Finally, the last section presents the integration of the Decomposition project into a library that aims to provide a solver of linear equations for systems originating from finite element computations.

2.1 Libraries Used

This section aims to introduce the two main libraries used in the *Decomposition* project. The first, *Template Numerical Library (TNL)*[1], provides this project with data structures and parallel functionalities. The second, *CUDA Libraries* (*cuSOLVER*[2], *cuBLAS*[3]), provides state-of-the-art decompositions and linear system solvers.

2.1.1 Template Numerical Library (TNL)

The Template Numerical Library (TNL) [39, 40] is an open-source and collaborative project licensed under the MIT license. According to the core team behind the project, TNL aims to be the C++ Standard Template Library (STL) for HPC [41]. In practice, this means facilitating a base for the development of, for example, efficient numerical solvers and other HPC algorithms. Similarly to STL, TNL is implemented in C++ and uses up-to-date programming paradigms to provide a user-friendly interface. Additionally - as the name suggests - TNL utilizes the advantages provided by C++ templates such as minimal overhead runtime and broad compatibility of functionalities. In terms of HPC, TNL provides a unified interface for managing multicore CPUs, GPUs, and distributed systems.

While TNL provides a wide range of data structures and functionalities, this section will only briefly present those utilized in the *Decomposition* project. First, a selection of data structures will

[1] TNL webpage URL: `https://tnl-project.org`
[2] cuSOLVER webpage URL: `https://developer.nvidia.com/cusolver`
[3] cuBLAS webpage URL: `https://developer.nvidia.com/cublas`

be introduced followed by parallel functionalities. See *Template Numerical Library User Guide* [42] for examples and tutorials concerning the presented content.

Data Structures

Array One of the most basic data structures is the `TNL::Containers::Array` class or `Array` for short [42]. An array type is declared using the following template parameters:

- `Value` - The array data type, e.g., `double`.
- `Device` - The device where an array operation is to be performed - either on the CPU or on the GPU, denoted using `TNL::Devices::Host` or `TNL::Devices::Cuda`, respectively.
- `Index` - The indexing data type, e.g., `int`.
- `Allocator` - The allocator type used for the allocation and deallocation of data. This parameter dictates the memory space the data is allocated in, e.g., on the host (CPU memory space), or the device (CUDA memory space). By default, the allocator corresponding to `Device` is set.

At its core, `Array` contains a pointer to the data and the size of the array. Additionally, methods providing different functionalities are present, for example, `resize()` or `forAllElements()`, where the latter executes a given lambda expression[4] for each element of the array; the lambda is executed by the same device whose memory space contains the data. Note that while all methods can be executed on the host (CPU), only those prefixed with `__cuda_callable__` can be executed on the device (Nvidia GPU). The `__cuda_callable__` macro is defined as `__device__ __host__`, where `__device__` indicates that the function can only be called from and executed on the device, and `__host__` indicates that the function can only be called from and executed by the host.

Vector Extending `Array` is the `TNL::Containers::Vector` class [42]. Its template parameters are identical to that of `Array` except for changes in nomenclature. While its parent class possesses functionalities largely centered around memory management, `Vector` adds basic vector operations such as addition, subtraction, scalar multiplication, etc.

Dense Matrix The data structure implementing a dense matrix, i.e., a matrix storing all its values (zeros and nonzeros inclusive), is the `TNL::Matrices::DenseMatrix` class [42]. The class type is declared using the following template parameters:

- `Real` - The type of the matrix's elements, e.g., `double`.
- `Device` - The device where the matrix is allocated. It can be either on the CPU or on the GPU denoted using `TNL::Devices::Host` or `TNL::Devices::Cuda`, respectively.
- `Index` - The indexing type of the matrix's elements, e.g., `int`.
- `Organization` - The ordering of matrix elements in the matrix's data vector. It can be either row-major or column-major order, denoted by `RowMajorOrder` or `ColumnMajorOrder`, respectively, found in the `TNL::Algorithms::Segments::ElementsOrganization` namespace.

[4] C++ lambda expressions on cppreference: `https://en.cppreference.com/w/cpp/language/lambda`

- `RealAllocator` - The allocator for the matrix's elements.

Analogous to `Vector`, `DenseMatrix` contains a variety of functionalities ranging from different constructors and data accessors to matrix operations and per-element methods.

In this project, an often-used method is the data-accessing operator: `operator( row, col )`. This operator returns a reference to an element at a given row and column. While this access is fast, it can only be used to access data from the same device whose memory space contains the data as it is prefixed with `__cuda_callable__`. For cross-device data access, either `setElement( row, col, value )` or `getElement( row, col )` can be used. Notwithstanding, the use of `setElement` and `getElement` is discouraged as copying a single element between devices is synonymous with low performance. Note that `getElement()` returns the value of the element.

The `DenseMatrix` data structure was chosen to represent the matrices in the implementation presented earlier as the focus of this project is the development of a dense decomposition algorithm.

View To comfortably use the aforementioned data structures in CUDA kernels, TNL provides an encapsulation mechanism without ownership in the form of *views*. For example, the `Array` class contains a view referred to as `ArrayView`. It can be assimilated to a pointer to an `Array` instance that retains only methods of `Array` that do not manage memory. Specifically, the `ArrayView` of an `Array` instance is bound to the `Array` instance's data via its data pointer, i.e., the data can be read and overwritten using the view but the array cannot be resized. Note that the destruction of an `ArrayView` instance does not affect the `Array` instance. Another important characteristic of `ArrayView` is that, unlike `Array`, its copy-constructor creates a shallow copy, i.e., only the pointer and size are changed. This allows for array views to be both passed by value to CUDA kernels and captured by value in lambda functions (including lambda functions executed on the device). In contrast, the overloaded operator `operator=` of both `Array` and `ArraView` performs a deep copy of the data.

Parallel Functionalities

Parallel For Loop Parallelization of a for loop performing independent tasks in CUDA is a simple task. However, for user comfort, TNL provides a parallel for-loop implementation: `TNL::Algorithms::parellelFor( begin, end, f )`, where `begin` and `end` specify the boundary indices, and `f` represents the lambda function performed in each iteration. Note that the implementation supports multi-index values, i.e., parallelization of nested for loops is implicitly supported. Notwithstanding dynamic parallelism, TNL's parallelized for-loop cannot be run within kernels.

Parallel Reduction Parallel reduction with sequential addressing, described in Section 1.2.5, is available in TNL in different forms. The basic form, where an array is reduced to a single value, is provided by `TNL::Algorithms::reduce( array, reduction )` where `array` can be an array, view, or another compatible object, and `reduction` specifies the reduction operation to perform. The reduction operations available in TNL are, for example, `TNL::Max{}`, `TNL::LogicalAnd{}`, etc. Additionally, TNL provides parallel reduction with an argument: `TNL::Algorithms::reduceWithArgument( array, reduction )`, which returns both the reduced element and its index. For this function, the `reduction` operation can be, e.g., `TNL::MaxWithArg{}`, and the returned `std::pair` contains the maximum value of the array and the index at which it is located. Moreover, both reduction functions provide customization

through additional parameters. For example, the `reduceWithArgument( begin, end, fetch, reduction, identity )` function takes the following parameters:

- `begin` - the index where the reduction begins;
- `end` - the index before which the reduction ends;
- `fetch` - the lambda function to fetch the input data;
- `reduction` - the lambda function to perform the reduction operation; and
- `identity` - the identity element.

2.1.2 CUDA Libraries

The two CUDA libraries used in the *Decomposition* project are *cuSOLVER* [43] and *cuBLAS* [44]. Both libraries were developed by Nvidia to provide users with an extensive collection of state-of-the-art functionalities that leverage the potential of Nvidia GPUs.

Similarly to the introduction of TNL in Section 2.1.1, only the functionalities utilized in the project will be briefly presented.

cuSOLVER

The cuSOLVER library provides functionalities for decomposing matrices and finding solutions to linear systems. While it can be argued that sparse matrices are more common in HPC, the library includes methods for both dense and sparse matrices.

As mentioned in Sections 1.3.3 and 2.1.1, the *Decomposition* project aims to present a novel approach to dense-matrix decomposition. Thus, to provide a comparison with an industry standard, the `cusolverDnXgetrf()` function was included in a TNL-compatible wrapper. The chosen function performs LU decomposition with partial pivoting:

$$\mathbf{PA} = \mathbf{LU}\,, \tag{2.1}$$

where **P** is a permutation matrix, **A** is a coefficient matrix, **L** is a unit lower-triangular matrix, and **U** is an upper-triangular matrix. In other words, unlike CM, the main diagonal consisting of ones is found in **L** and not in **U**.

The function is data-agnostic, i.e., both single and double precision can be used. For completeness, note that the function's name follows cuSOLVER's naming conventions `cusolverDn<t><operation>` [43], where:

- `t` represents the data type, for example, `S`, `D`, `C`, `Z`, or `X`, i.e., `float`, `double`, `cuComplex`, `cuDoubleComplex`, or the generic type, respectively; and
- `operation` represents the factorization operation, for example, `potrf` (Cholesky factorization), `getrf` (LU with partial pivoting), `geqrf` (QR factorization), or `sytrf` (Bunch-Kaufman factorization).

While the `cusolverDnXgetrf()` function takes many parameters, only a select few will be shown in Listing 2.1.

```
cusolverStatus_t cusolverDnXgetrf(
  int64_t m,                // Number of rows in matrix A
  int64_t n,                // Number of columns in matrix A
  cudaDataType dataTypeA,   // The element type of matrix A, e.g., double
  void *A,                  // Pointer to GPU memory where matrix A is stored as an array in column-major order
  int64_t *ipiv )           // Pointer to GPU memory where the pivoting vector is stored; if set to nullptr, then partial pivoting is not performed
```

Listing 2.1: The function declaration of `cusolverDnXgetrf()` with a selection of parameters.

On output, the elements in the lower triangle of matrix `A` are those of matrix **L**, and the elements in the upper triangle of `A` - including the main diagonal - are those of **U**:

$$\mathtt{A} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ l_{2,1} & u_{2,2} & u_{2,3} & \dots & u_{2,n} \\ l_{3,1} & l_{3,2} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & u_{n-1,n} \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & u_{n,n} \end{bmatrix}. \tag{2.2}$$

As the method exhibits a characteristic where the values of the main diagonal of matrix **L** are equal to 1, irrespective of the input matrix, the main diagonal of **L** is not stored.

For the full list of parameters, see the function's documentation[5] [43]. Additionally, an example where the `cusolverDnXgetrf()` function is used can be found in `cuSOLVER/Xgetrf/cusolver_Xgetrf_example.cu` in the *CUDALibrarySamples* GitHub repository [45].

To provide a complete linear solver, along with `cusolverDnXgetrf()`, the cuSOLVER library also provides the `cusolverDnXgetrs()` function. Note that the function is only responsible for solving a linear system with multiple right-hand sides [43]. Similarly to the aforementioned cuSOLVER function, `cusolverDnXgetrs()` is data-agnostic. For a selection of input parameters for `cusolverDnXgetrs()` see Listing 2.2.

```
cusolverStatus_t cusolverDnXgetrs(
  int64_t n,                // Number of rows/columns in matrix A
  int64_t nrhs,             // Number of right-hand sides of the linear system, i.e., the number of columns in matrix B
  cudaDataType dataTypeA,   // The element type of matrix A, e.g., double
  const void *A,            // Pointer to GPU memory where matrix A is stored as an array in column-major order
  const int64_t *ipiv,      // Pointer to GPU memory where the pivoting vector is stored; if set to nullptr, then partial pivoting is ignored
  cudaDataType dataTypeB,   // The element type of matrix B, e.g., double
  void *B )                 // Pointer to GPU memory where matrix B is stored as an array in column-major order
```

Listing 2.2: The function declaration of `cusolverDnXgetrs()` with a selection of parameters. Note that matrix `A` should adhere to the format of the matrix produced by `cusolverDnXgetrf()` [43].

[5] Documentation of `cusolverDnXgetrf()`: `https://docs.nvidia.com/cuda/archive/12.0.0/cusolver/index.html#cusolverdnxgetrf`

Similarly to `cusolverDnXgetrf()`, the full list of parameters can be found in the documentation of `cusolverDnXgetrs()`[6] [43] and an example where the function is used can be found in `cuSOLVER/Xgetrf/cusolver_Xgetrf_example.cu` in the *CUDALibrarySamples* GitHub repository [45].

cuBLAS

The cuBLAS library serves as a CUDA-specific Linear Algebra Subroutine Library [44]. Given that the *Decomposition* project aims to provide a complete, TNL-compatible solution for LU decomposition and for solving systems of equations, the `cusolverDnXgetrs()` function alone is not sufficient. While it is a performant solver, according to the cuSOLVER documentation [43], it is only compatible with the matrix format produced by `cusolverDnXgetrf()`. In other words, `cusolverDnXgetrs()` is only capable of solving systems:

$$\mathbf{LUX} = \mathbf{PB}\,, \tag{2.3}$$

where **L** is a *unit lower*-triangular matrix and **U** is an upper-triangular matrix. Note that, in practice, matrices **L** and **U** are stored together in a single matrix, following the format shown in Equation 2.2. Since CM can only produce the opposite, i.e., a lower-triangular matrix coupled with a *unit upper*-triangular matrix, an alternative approach was needed for solving linear systems using CM.

One such alternative, offered by the cuBLAS library, is the `cublas<t>trsm()` function. Similarly to the cuSOLVER functions introduced earlier, `t` can be either `S`, `D`, `C`, or `Z`, i.e., `float`, `double`, `cuComplex`, or `cuDoubleComplex`, respectively.

However, unlike `cusolverDnXgetrs()`, `cublas<t>trsm()` is only capable of solving *triangular* linear systems with multiple right-hand sides. To use it for solving linear systems with LU decomposition, it must be called twice (as detailed in Step 2 of the two-step process for solving a system of linear equations mentioned in Section 1.3.1):

1. To solve $\mathbf{LY} = \mathbf{PB}$, where **Y** and **B** are appropriately-sized matrices.
2. To solve $\mathbf{UX} = \mathbf{Y}$, where **X** is an appropriately-sized matrix.

For this purpose, the cuBLAS datatype `cublasFillMode_t` can be used in combination with `cublasDiagType_t` to indicate which part of the matrix should be used by the `cublas<t>trsm()` function. The `cublasFillMode_t` datatype can be either `CUBLAS_FILL_MODE_LOWER`, `CUBLAS_FILL_MODE_UPPER`, or `CUBLAS_FILL_MODE_FULL`. The first two options represent the cases when either the lower triangle or the upper triangle of the matrix is to be used, respectively, while the last option represents the case when the entire matrix is to be used. The `cublasDiagType_t` datatype can be either `CUBLAS_DIAG_NON_UNIT` or `CUBLAS_DIAG_UNIT`. The former represents the case when the main diagonal of the matrix does not consist of ones, whereas the latter represents the case when it does.
For clarity, to solve $\mathbf{LY} = \mathbf{PB}$ using `cublas<t>trsm()`, the function must be called with the following parameters:

[6] Documentation of `cusolverDnXgetrs()`: `https://docs.nvidia.com/cuda/archive/12.0.0/cusolver/index.html#cusolverdnxgetrs`

- `CUBLAS_FILL_MODE_LOWER` and `CUBLAS_DIAG_UNIT` when **L** is a *unit* lower-triangular matrix; and
- `CUBLAS_FILL_MODE_LOWER` and `CUBLAS_DIAG_NON_UNIT` when **L** is a lower-triangular matrix.

The parameters are set similarly for **U** when solving **UX** = **Y**, with the exception of using `CUBLAS_FILL_MODE_UPPER` instead of `CUBLAS_FILL_MODE_LOWER`.

For a selection of input parameters for `cublas<t>trsm()` see Listing 2.3.

```
cublasStatus_t cublasDtrsm(
  cublasFillMode_t uplo, // Indicator of which part of the matrix is to be used
  cublasDiagType_t diag, // Indicator signifying whether the matrix has a unit main diagonal
  int m,                 // Number of rows of matrix B and rows/columns of matrix A
  int n,                 // Number of columns of matrix B
  const double *A,       // Pointer to GPU memory where matrix A is stored as an array in column-major order
  double *B )            // Pointer to GPU memory where matrix B is stored as an array in column-major order
```

Listing 2.3: The function declaration of `cublasDtrsm()` with a selection of parameters.

The full list of parameters can be found in the documentation of `cublas<t>trsm()`[7] [44].

In summary, `cublas<t>trsm()` can be used to solve linear systems regardless of the LU matrix format.

2.2 Decomposition Project

This section aims to present the *Decomposition* project, which houses a selection of LU decomposition and linear-system-solving algorithms. It is important to mention that the initial version of the Decomposition project was conceptualized and implemented as part of *Parallel LU Decomposition for the GPU* [2]. In the context of this thesis, the project was refactored and extended to include several features. Therefore, for completeness, the project will be presented in its entirety with occasional references instead of detailed explanations.

The development of the project was tracked in the Decomposition repository on GitLab[8]. Note that access to the repository is restricted to members of the TNL group[9]. However, the project is available on request or as an attachment to this thesis.

The project is made up of the following parts:

- Unit tests - The tests are written using *GoogleTest*, which is Google's C++ testing framework[10].

[7] Documentation of `cublas<t>trsm()`: `https://docs.nvidia.com/cuda/archive/12.0.0/cublas/index.html#cublas-t-trsm`

[8] Decomposition GitLab repository URL: `https://gitlab.com/tnl-project/decomposition`

[9] TNL GitLab group URL: `https://gitlab.com/tnl-project`

[10] GoogleTest GitHub repository URL: `https://github.com/google/googletest`

- Implementations of algorithms - The project is primarily written in C++, except for CUDA-extended C++ which is used in the implementations of algorithms.
- Benchmarks - The benchmarks are written in C++, with additional components such as benchmark-running scripts written in Bash[11], and benchmark-result-visualizing scripts written in Python[12].

As mentioned in *Parallel LU Decomposition for the GPU* [2], the Decomposition project was not developed as part of TNL. However, TNL's project structure and building processes were adopted and tailored to fit the requirements of this project. The main reason behind this decision was that several concepts in TNL could be easily reused. Furthermore, it allows for the project to be incorporated into TNL in the future. For more information regarding the building process, refer to the project's README file[13].

First, the unit tests will be briefly presented. Then, the LU decomposition and linear-system-solving solutions implemented in the project will be detailed. The final part will introduce the benchmarks of the implementations.

2.2.1 Unit Tests

The development of the Decomposition project was a continuous loop of theorizing, implementing, and testing. Therefore, unit tests played a crucial role in facilitating smooth development. While the structure of unit tests was adopted from TNL, certain modifications were made to meet the requirements of the implementations [2]. Specifically, the unit tests needed to fulfill the following requirements:

1. Lightweight - The unit tests are executed with every compilation of the project and, therefore, should be executed quickly.
2. Reusable - The same test should be applicable to different algorithms.
3. Versatile - The same test should be capable of running with different data types.
4. Thorough - Certain implementations had multiple potential points of failure that must be reliable.

To satisfy the requirements posed above, the following concepts were used and implemented:

- Sample problems - A set of problems that can be used in any test. For example, matrix `A` along with its correct decomposition, or a system of linear equations in matrix form along with its solution. To satisfy the "lightweight" requirement, the dimensions of matrices in the problems were limited to a range from 2-by-2 to 38-by-38.
- Parametrized tests - The test methods were parametrized with template parameters provided by GoogleTest. These template parameters are used to pass, for example, the type of matrix with different data types or the algorithm used in the test.

[11] Bash website URL: `https://www.gnu.org/software/bash`

[12] Python website URL: https://www.python.org

[13] Decomposition project's README file: `https://gitlab.com/tnl-project/decomposition/-/blob/master/README.md`

- Configurable result verification - Due to the nature of some algorithms, the results produced by their implementations are not as accurate as others. Thus, certain tests have to expect results within a range.
- Point-of-failure testing - Using GoogleTest, it is possible to thoroughly verify the points of failure in each implementation. This includes validating the type of exception thrown and examining the contents of the exception's message.

Furthermore, inspired by TNL and to ensure that the unit tests covered the desired functionalities, code coverage was incorporated into the project using a tool called *LTP GCOV extension (LCOV)* [46]. As of 9th June 2023, the line coverage was 90.6% and the function coverage was 76.6%. Note that the coverage percentages may be affected by the fact that CUDA kernels were not recognized as "hit" during execution, even if their calls were.
The full coverage report is available on request or as an attachment to this thesis.

2.2.2 Implemented Algorithms

The core implementations in the project are divided into two groups: *Decomposers* and *Solvers*. The *Decomposers* group consists of implementations that perform LU decomposition with and without partial pivoting. The *Solvers* group consists of implementations that use the output of an LU decomposition algorithm to solve linear systems with and without partial pivoting.

In the context of this project, an implementation from the *Decomposers* group is referred to as a *decomposer*, and an implementation from the *Solvers* group is referred to as a *solver*.

Decomposers

In particular, the *Decomposers* group consists of implementations that perform LUP, i.e., **PA** = **LU**, or LU decomposition without partial pivoting, i.e., **A** = **LU**. The full list of decomposers and their characteristics is presented in Table 2.1.

Decomposer	With / Without PP	Unit diag. in	Device supported
CM(PP)	Yes / Yes	**U**	CPU
CuSolverDnXgetrf(PP)	Yes / Yes	**L**	GPU
GEM	Yes / No	**L**	CPU
ICM_*x*(PP)	Yes / Yes	**U**	CPU*, GPU
PCM_*x*(PP)	Yes	**U**	GPU

Table 2.1: The decomposers made available by the Decomposition project. The decomposers highlighted in gray are discussed in this thesis, and those in **bold** font were implemented by the author of the project. The "PP" suffix indicates whether partial pivoting is used. The "*x*" in the name of certain decomposers signifies their CUDA thread configuration. The "Unit diag. In" column indicates which of the two matrices, **L** or **U**, contains the unit diagonal on output. Note that ICM_*x*PP is only implemented on the GPU as the CPU implementation would be highly inefficient.

As can be seen from Table 2.1, the Decomposition project includes the implementation of the Gaussian Elimination Method (GEM) decomposer. However, it will not be a subject of discussion in this thesis as it was primarily used for verifying the accuracy of other decomposers. Its use is discouraged for any purpose other than the verification of accuracy.

Next, the implementations of the selected decomposers will be presented.

Crout's Method with Partial Pivoting (CMPP) The sequential algorithm of CMPP, detailed in Section 1.3.2, was implemented only for the CPU since it would not be efficient on the GPU without any modifications. The declaration of the `decompose()` method for the CMPP decomposer is shown in Listing 2.4, and the definition can be found in Appendix A. The definition of `decompose()` differs from the algorithm described in Section 1.3.2 in the tolerance of partial pivoting. In this context, *tolerance* refers to the conditions under which a row is pivoted. While most LUP algorithms have no tolerance, i.e., they pivot every row, the CMPP decomposer only pivots a row if its diagonal element, $l_{j,j}$, is smaller in absolute value than (or equal to) a specified threshold, e.g., 1e-5. In the context of this project, this behavior is referred to as *conditional partial pivoting*. Initially, conditional partial pivoting, described in Step 4 of ICMPP's algorithm on page 36, was included in the CMPP decomposer to test the stability of the concept. However, it remained as further testing showed promising results.

```
1 template< typename Matrix, typename Vector >
2 static void
3 decompose( Matrix& LU, Vector& piv );
```

Listing 2.4: The declaration of the `decompose()` method for the CMPP decomposer. On input, matrix `LU` is assumed to contain the values of **A**, and `piv` is expected to be appropriately sized. On output, matrix `LU` contains the values of matrices **L** and **U** in the format presented in Equation 2.4, and `piv` contains the row permutations in the format set by cuSOLVER and cuBLAS, i.e., row `i` was swapped with row `piv[i]` [43]. The template parameters, `Matrix` and `Vector`, are expected to be data structures from TNL that inherit from the `TNL::Matrices::DenseMatrix` and `TNL::Containers::Vector` types, respectively.

$$\texttt{LU} = \begin{bmatrix} l_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ l_{2,1} & l_{2,2} & u_{2,3} & \dots & u_{2,n} \\ l_{3,1} & l_{3,2} & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & u_{n-1,n} \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & l_{n,n} \end{bmatrix}. \tag{2.4}$$

Similarly to cuSOLVER, `piv` can contain any values on input as they are all replaced during computation. However, in the case of the CMPP decomposer, the values of `piv` are initialized to their default values within the `decompose()` method prior to the main computation. Specifically, the default values follow the format set by cuSOLVER and cuBLAS, i.e., `piv[0] = 1`, `piv[1] = 2`, etc. Note that the values of `piv` must be set to their default pivoting values by every decomposer that uses conditional partial pivoting. The reasoning behind this requirement stems from the fact that conditional partial pivoting does not attempt to pivot all rows.

Parallel Crout's Method with Partial Pivoting (PCM_*x*PP) Originally, the PCM_*x*PP decomposer was conceived as a GPU-compatible alternative to CMPP that preserves its accuracy as a direct method. However, it remained in the project as its performance was often comparable to that of ICM_*x*PP. The algorithm of PCM_*x*PP is identical to that of CMPP except for minor parallelizations. Specifically, the algorithm consists of the following steps (changes compared to the algorithm of CMPP are highlighted in green):

1. Set j equal to 1.
2. If j is greater than n, then stop the execution as matrices **L** and **U** have been successfully computed.

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

4. Pivot row j:

   (a) In the values computed in step 3, find the index (p) of the element largest in absolute value using parallel reduction:

   $$p = \arg\max_k \left\{ \left| l_{k,j} \right| : k = j, \ldots, n \right\} . \tag{2.5}$$

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

   (c) 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 $\mathbf{U}$ in parallel.

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

The simultaneous computations mentioned in Steps 3 and 5 are possible due to the dependency of elements mentioned in Section 1.3.2 and shown in Figure 1.13. For example, to compute the element $l_{j,j}$, elements from $l_{j,1}$ to $l_{j,j-1}$ and from $u_{1,j}$ to $u_{j-1,j}$ are required. To compute the element $l_{j+1,j}$, elements from $l_{j+1,1}$ to $l_{j+1,j-1}$ and from $u_{1,j}$ to $u_{j-1,j}$ are required. In other words, to compute an element in $\mathbf{L}$, only the elements to the left in its row and above the main diagonal in its column are required. Therefore, the computation of each element in Step 3 does not depend on the computation of other elements in the same step. Thus, values in Step 3 can be computed in parallel. The parallelization of the computation in Step 5 follows the same logic.

In Step 4a, `TNL::Algorithms::reduceWithArgument` (introduced in Section 2.1.1) was used to perform the parallel reduction.

However, the approach taken to parallelize the computation in Steps 3 and 5 is not highly parallel, i.e., the number of elements that can be computed simultaneously is low. In particular, the maximum number of elements that can be computed in parallel is equal to the number of rows/columns of the input matrix. Thus, the maximum number of active threads at a point in time is equal to the number of rows/columns. This lack of parallelism reflects itself on the implementation of the algorithm as it comprises the CPU sequentially tasking the GPU to perform simultaneous computations.

The declaration of the `decompose()` method for the PCM_*x*PP decomposer is identical to that of the CMPP decomposer (shown in Listing 2.4), and the definition can be found in Appendix B. Note that, unlike `LU`, `piv` is assumed to be allocated on the Host as it is not used in kernels and its values are set by the CPU.

In the caption of Table 2.1, it was mentioned that the x in PCM_*x*PP signifies the CUDA thread configuration of the decomposer. Specifically, the number of threads in each one-dimensional CUDA thread block is defined as x^2.

Iterative Crout's Method with Partial Pivoting (ICM_*x*PP) One of the main tasks for this thesis was to implement an LU decomposition algorithm with partial pivoting (LUP) for the GPU. As part of *Parallel LU Decomposition for the GPU* [2], LU decomposition (without partial pivoting) was implemented both for the CPU and for the GPU in the form of ICM*x*. However, the performance of the CPU implementation was low, and it served solely as an initial proof-of-concept that was neither developed nor optimized further. Therefore, in the context of this thesis, the ICM_*x*PP algorithm, introduced in Section 1.3.3, was implemented only for the GPU, as specified in Table 2.1.

While the implementation of ICM_*x*PP was optimized throughout the development of the project, only the final version will be detailed in this thesis.

The GPU implementation of ICM_*x*PP follows the algorithm introduced in Section 1.3.3 with minor modifications. Therefore, its description is split into three parts:

1. Initial estimate of the decomposition
2. Processing by sections
3. Computing one iteration of a section

Initial Estimate of the Decomposition The first step of the ICMPP algorithm, introduced in Section 1.3.3, is to estimate the decomposition of matrix **A**, i.e., to estimate matrices **L** and **U**. This estimate is then improved in every iteration of the algorithm until the change between iterations is smaller than some tolerance. In the context of this project, once the change in values of a matrix between iterations is smaller than a set tolerance, the matrix is declared as *processed*. Thus, the choice of the initial estimate can have a significant impact on the performance and accuracy of the decomposer.

The default choice, as shown in the aforementioned algorithm, is to estimate matrices **L** and **U** using **A** in the following manner:

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

The method declaration providing this functionality is identical to the one shown in Listing 2.4.

However, to allow for experimentation, the ICM_*x*PP decomposer also provides a method that allows the user to supply an initial estimate. This feature is facilitated via an overloaded method whose declaration is shown in Listing 2.5.

```
template< typename Matrix, typename Vector >
static void
decompose( Matrix& A, Matrix& LU, Vector& piv );
```

Listing 2.5: The declaration of the overloaded `decompose()` method for the ICM_*x*PP decomposer that allows the user to supply an initial estimate of the decomposition. On input, matrix `A` is assumed to contain the values of **A**, matrix `LU` is assumed to contain the initial estimate of the decomposition, and `piv` is expected to be appropriately sized and allocated on the Host. On output, matrix `LU` contains the values of matrices **L** and **U** in the format presented in Equation 2.4, and `piv` contains the row permutations.

This method will be used in the descriptions that follow.

Processing by Sections The ICMPP algorithm, introduced in Section 1.3.3, suggests that in every iteration, all values in matrix `LU` are to be computed. However, the decomposition of a matrix can require thousands of iterations. Moreover, to decompose an $n \times n$ matrix, such an approach would require n^2 elements to be computed in every iteration. Assuming that each CUDA thread was to

compute one element in an iteration, this would translate to n^2 threads being allocated in every iteration. This approach does not scale well.

Furthermore, due to the dependency of elements described in Section 1.3.2, it is futile to compute values of `LU` that have extensive chains of dependencies if the elements they depend on have not been processed. The chains of dependencies for a specific element are visualized in Figure 2.1.

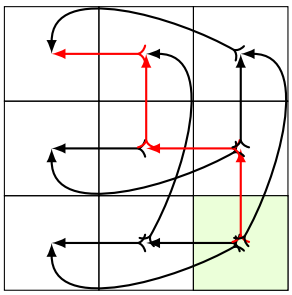

Figure 2.1: Visualization of the chains of dependencies for element `LU[ 2 ][ 2 ]` (green box) in a 3-by-3 matrix `LU`. Each arrow points to the element that the element it originates from requires for its computation. For example, to compute `LU[ 2 ][ 2 ]`, the elements it points to are required, i.e., `LU[ 2 ][ 0 ]`, `LU[ 2 ][ 1 ]`, `LU[ 1 ][ 2 ]`, and `LU[ 0 ][ 2 ]` are required. Those elements then depend on other elements, etc. The red arrow represents the longest chain.

From Figure 2.1, it can be seen that `LU[ 2 ][ 2 ]` depends on numerous elements. If the elements on which `LU[ 2 ][ 2 ]` depends are of low quality and are used in its computation, then its value will also be low quality. In this context, the term *quality of an element* refers to how close the element is to its final value. In other words, how close it is to being processed. For example, low quality indicates that the element is far from its final value, i.e., it is not yet processed.
Therefore, if `LU[ 2 ][ 2 ]` is computed early in the iteration process, its value is of no use, and resources will have been wasted to compute it.

In the context of the entire matrix, the values in the top-left corner of `LU` are processed first since their chains of dependencies are negligible. As a result, the elements that depend on them are processed in a few iterations, and so on. In other words, originating in the matrix's top-left corner, there is an "avalanche" effect that eventually results in the processing of elements in the bottom-right corner of the matrix. However, this suggests that computing elements far from the quality-increasing avalanche is futile. Therefore, it is more efficient to compute them only when the elements they depend on are of high quality.

To address these issues, a modified approach is used where, in each iteration, only a specific part of the matrix is computed instead of the entire matrix. This is achieved by dividing the matrix into square *sections* of equal size. The sections are then processed in a specific order. Adhering to:

- the dependency of elements depicted in Figure 1.13 of Section 1.3.2, and
- the order of computation described in the ICMPP algorithm introduced in Section 1.3.3,

the sections of matrix `LU` are processed in the order shown in Figure 2.2.

Before outlining the concept of *processing by sections*, it is necessary to define a term used thoroughly in this thesis: the *bad element*. In this context, the term *bad element* refers to an element located on the main diagonal of a matrix that breaks the tolerance of conditional partial pivoting. In

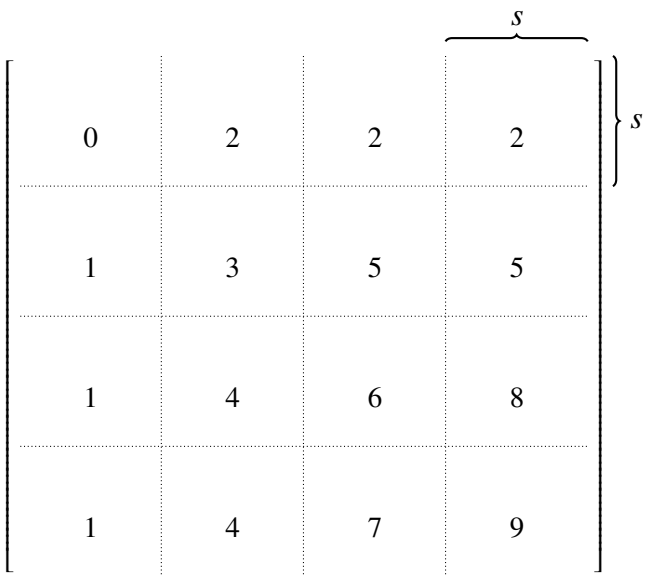

Figure 2.2: Visualization of matrix `LU` divided into sections of size $s \times s$. The number inside each section indicates the order in which it is processed. First, the top-left section, labeled as section "0", is processed. Then, the sections below it, i.e., sections labeled with a "1", are processed in parallel. Subsequently, the sections to the right of section "0" are processed in parallel, etc.

other words, it is an element on the main diagonal whose absolute value is smaller than or equal to a set tolerance, e.g., 1e-5.

For clarity, the outline of processing the matrix by sections using the ICMPP algorithm is presented below:

1. Start with the top-left diagonal section.
2. Set the index from which a bad element will be searched for to the first element in the main diagonal of the diagonal section; the index is stored in the `badEl_searchStart` variable.
3. In the diagonal section, starting at the index `badEl_searchStart`, find the first bad element. If no bad element is found, the index is set to an illegal value to indicate the absence of a bad element.
4. Process the diagonal section:
   (a) Perform one iteration on the values in the diagonal section. However, if a bad element exists, only compute values that are not located to the bottom-right of the bad element. In other words, do not compute values whose indices fulfill the condition: `row >= badEl_rowcol && col > badEl_rowcol`, where `badEl_rowcol` is the row and column index of the bad element.
   (b) Check if a bad element is present between `badEl_searchStart` and the index of the current bad element. If it is, then overwrite the index of the current bad element with its index.
   (c) If the diagonal section is processed, proceed to Step 5. Otherwise, return to Step 4a.
5. If there are no sections below the diagonal section, then the decomposition is complete.

6. Process the sections below the diagonal section. However, if a bad element exists, then only compute values up to and including the column containing the bad element.
7. If a bad element exists, then pivot the row containing the bad element. If the new value is equal to zero then the decomposition has failed. If the new value is nonzero but breaks the pivoting tolerance, then set `badEl_searchStart` to the next element on the diagonal so the computation can continue. In other words, by incrementing `badEl_searchStart`, the bad element is avoided in the following searches.
8. If the index of the bad element is smaller than the index of the bottom-right-most element of the diagonal section, then return to Step 3.
9. Process the sections to the right of the diagonal section.
10. Move to the next diagonal section and proceed to Step 2.

Next, a selection of excerpts from the implementation of ICM_*x*PP will be described. For the complete definition of the `decompose()` method for the ICM_*x*PP decomposer, see Appendix C.

Figure 2.2 shows matrix `LU` divided into square equally-sized parts referred to as *sections*. The size of a section is determined by the dimensions of matrix `LU`, as shown in Listing 2.6.

```
1  template< const int BLOCK_SIZE >
2  template< typename Matrix, typename Vector >
3  void
4  IterativeCroutMethod< BLOCK_SIZE >::decompose( Matrix& A, Matrix& LU, Vector&↩
       piv )
5  {
6    using Index = typename Matrix::IndexType;
7
8    const Index num_rows = LU.getRows();
9    const Index num_cols = LU.getColumns();
10
11   Index sec_size = min( max( num_cols / 10, (Index) 256 ), (Index) 1024 );
12   sec_size = ( sec_size + BLOCK_SIZE - 1 ) / BLOCK_SIZE * BLOCK_SIZE;
13
14   // ...
15 }
```

Listing 2.6: An excerpt from the definition of the overloaded `decompose()` method for the ICM_*x*PP decomposer. This definition matches the declaration shown in Listing 2.5. The `Index` type and dimension-representing variables are included for clarity. The template parameter `BLOCK_SIZE`, equivalent to x in ICM_*x*PP, represents the number of threads in the 1st and 2nd dimensions of a CUDA thread block.

Initially, the size of a section is set to $1/10$ of the dimensions of `LU`. However, the value is limited to the interval $[256, 1024]$, i.e., if the size of a section exceeds either boundary of the interval, it is adjusted to the value of the nearest boundary. Then, the value is rounded to the nearest multiple of `BLOCK_SIZE` (8, 16, or 32) that is greater than or equal to it. For example, if `LU` is a 3,000-by-3,000 matrix, then the size of a section will be 300×300. However, if `LU` is a 12,000-by-12,000 matrix, then the size of each section is adjusted to 1024×1024.
The interval was determined as a result of extensive testing performed as part of *Parallel LU Decomposition for the GPU* [2]. For clarity, the value is rounded to assure correct loop unrolling with shared memory in the computing kernels. For a thorough explanation, see the last paragraph of Section 2.1 in *Parallel LU Decomposition for the GPU* [2].

The next excerpt, shown in Listing 2.7, highlights the processing of a diagonal section.

```
// ...
using Real = typename Matrix::RealType;
// Matrix representing LU in the next iteration
Matrix LUnext;
LUnext.setLike( LU );

// Processing tolerance
const Real process_tol = 0.0;
// Pivoting tolerance
const Real piv_tol = 1.0e-5;

// CUDA grid configuration
Index blocks = sec_size / BLOCK_SIZE;
dim3 threads_perBlock( BLOCK_SIZE, BLOCK_SIZE );
dim3 blocks_perGrid( blocks, blocks );

// Flag to indicate that the diagonal section has been processed
TNL::Containers::Array< bool, TNL::Devices::Cuda > processed{ 1, false };

// ...

// Views are used to access the data
auto A_view = A.getView();
auto LU_view = LU.getView();
auto LUnext_view = LUnext.getView();
auto processed_view = processed.getView();

// Lambda for fetching the indices of bad elements
auto get_badEl_colIdxs = [ = ] __cuda_callable__( Index col ) -> Index
{
  return abs( LU_view( col, col ) ) <= piv_tol ? col : num_cols;
};

// Diagonal section start and end
Index dSec_start, dSec_end;
// Row/Column index of the bad element
Index badEl_idx;

// Loop through the diagonal sections
for( dSec_start = 0, dSec_end = min( num_cols, sec_size ); dSec_start < dSec_end; dSec_start += sec_size, dSec_end = min( num_cols, dSec_end + sec_size ) )
{
  // Set the starting point of the search for the first bad element
  Index badEl_searchStart = dSec_start;

  do { // Process the diagonal section and the sections below it
    // Get next badEl_idx
    badEl_idx = TNL::Algorithms::reduce< TNL::Devices::Cuda >( badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, num_cols );

    do {  // Process the diagonal section
      // Reset the processed flag
      processed_view.setElement( 0, true );

      // Compute the values up to and including the bad element
      DSecCompute_kernel< BLOCK_SIZE ><<< blocks_perGrid, threads_perBlock >>>( A_view, LU_view, LUnext_view, dSec_start, dSec_end, dSec_start, dSec_end, process_tol, processed_view, badEl_idx );

```

```
56        // Assign the values computed for the next iteration
57        DSecAssign_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, ↩
              LUnext_view, dSec_start, dSec_end, dSec_start, dSec_end, badEl_idx ↩
              );
58
59        // Check if a bad element is present between the previous bad element ↩
              and the current bad element
60        Index badEl_idx_new = TNL::Algorithms::reduce< TNL::Devices::Cuda >( ↩
              badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, ↩
              num_cols );
61        // If a bad element was detected before the current one, then set it as↩
               the new bad element
62        if( badEl_idx_new < badEl_idx )
63          badEl_idx = badEl_idx_new;
64      } while( ! processed_view.getElement( 0 ) );
65
66      // ... Processing of the sections below the diagonal section
67      // ... Pivoting of the bad element
68    } while( badEl_idx < dSec_end - 1 );
69    // ... Processing of the sections to the right of the diagonal section
70  }
```

Listing 2.7: An excerpt from the definition of the overloaded `decompose()` method for the ICM_*x*PP decomposer. The excerpt highlights how the processing of a diagonal section is implemented. For clarity, variables used in the code are declared and documented. The `DSecCompute_kernel()` and `DSecAssign_kernel()` kernels compute and assign values of the diagonal section, respectively. The kernels are presented separately in Listings 2.9 and 2.10.

The code in Listing 2.7 corresponds to the instructions from Step 1 to Step 5 in the outline presented earlier.
To find a bad element in the diagonal section, `TNL::Algorithms::reduce()` is used on Lines 47 and 60. The parameters of the variant of `reduce()` used correspond to the parameters of the `reduceWithArgument()` function presented in Section 2.1.1. For clarity, the lambda function defined on Line 29 fetches elements on the main diagonal from the `begin` index to the `end` index based on a condition. If the element at column `col` of the main diagonal is bad, then its column index is returned. In the opposite case, the size of the matrix is returned as the illegal value. Finally, the reduction operation, `TNL::Min`, returns the smallest index, i.e., the index of the first bad element on the main diagonal of the diagonal section.

Then, the diagonal section is processed as described in Step 4. Note that the processing flag `processed` is a one-element array containing a boolean that is used to indicate whether the diagonal section is processed. Line 51 shows that with the start of every loop, its value is set to `true`. Then, its view (`ArrayView` - explained in Section 2.1.1) is passed to the kernel where each thread evaluates whether the element it computed is processed or not. If it is not, then the thread sets the value of `processed` to `false`. While, generally, it is not good practice for multiple threads to write in an undefined order to the same memory address, in this case, it does not matter. The reasoning behind this statement is that the value written by all threads is the same. Furthermore, when the value of `processed` is read in the condition of the while loop on Line 64, it is done so only after the kernels have been executed. This is due to the fact that, as mentioned in Sections 1.2.2 and 1.2.4, tasks issued via the default stream are executed serially. In other words, the kernels are asynchronously called from the host, executed on the device in the order they were called, and then the value of `processed` is copied from the device to the host.

Another important aspect of the implementation is the processing tolerance, `process_tol`. As can be seen on Line 8 in Listing 2.7, it is set to `0.0`. The value was chosen to provide accurate

results as even small nonzero values, e.g., 1e-10, could greatly reduce the accuracy while increasing the performance by a negligible amount. The reason behind the great reduction in accuracy stems from the "avalanche" effect described earlier in *Processing by Sections*. However, in this case, the avalanche exhibits a quality-decreasing characteristic since the elements in the top-left corner would not be of the highest quality due to the relaxed tolerance. In other words, the imperfectness of elements in the top-left corner causes a butterfly effect resulting in highly inaccurate results. For more details, see Sections 2.2.1 and 2.3, and Chapter 3 in *Parallel LU Decomposition for the GPU* [2].

Once the values of the diagonal section have been processed either up to and including the bad element, or all, then the processing of the sections below the diagonal section begins. Once they are processed, the bad element can be pivoted. This order of operations assures that the elements below the bad element are processed when it needs to be replaced. Listing 2.8 shows the processing of the lower sections and the pivoting of the bad element. Note that the implementation of the pivoting function is shown in Listing C.2.

```
// ...
// Allocate and initialize an array of stream handles to process the sections in parallel
const Index nonDiagSecs_perRow = TNL::ceil( (double) num_cols / (double) sec_size ) - 1;
auto* streams = (cudaStream_t*) malloc( nonDiagSecs_perRow * sizeof( cudaStream_t ) );
for( Index i = 0; i < nonDiagSecs_perRow; ++i )
  cudaStreamCreate( &( streams[ i ] ) );

// Flags indicating the processing status of non-diagonal sections for the device and the host
TNL::Containers::Array< bool, TNL::Devices::Cuda, Index > nonDiagSec_processed{ nonDiagSecs_perRow, false };
// The host array is used to avoid copying individual processing variables between the host and the device
TNL::Containers::Array< bool, TNL::Devices::Host, Index > nonDiagSec_processed_host{ nonDiagSecs_perRow, false };
// ...
auto nonDiagSec_processed_view = nonDiagSec_processed.getView();

// Fill the pivoting vector with increments of 1 starting from 1 to num_rows.
BaseDecomposer::setDefaultPivotingValues( piv );
auto piv_view = piv.getView();
// ...

// Non-diagonal section start, end, and id (used to access the streams)
Index sec_start, sec_end, sec_id;

// Loop through the diagonal sections
for( dSec_start = 0, dSec_end = min( num_cols, sec_size ); dSec_start < dSec_end; dSec_start += sec_size, dSec_end = min( num_cols, dSec_end + sec_size ) )
{
  do { // Process the diagonal section and the sections below it
    // ... Processing of the diagonal section

    // The Diagonal section contains processed values - excluding the values to the bottom-right of the bad element
    // Compute the lower sections up to and including the column containing the bad element

```

```
32      // Limit the number of threads used based on the number of columns that will be computed
33      Index badEl_idx_cutoff = min( badEl_idx + 1, dSec_end );
34      Index lSec_width_rounded = ( badEl_idx_cutoff - dSec_start + BLOCK_SIZE - 1 ) & -BLOCK_SIZE;
35      dim3 lSec_blockPerGrid( TNL::max( lSec_width_rounded / BLOCK_SIZE, (Index) 1 ), blocks );
36
37      // Default to false so that all kernels are run in the first iteration
38      nonDiagSec_processed_view.setValue( false );
39
40      do { // Process the lower sections in parallel
41        nonDiagSec_processed_host = nonDiagSec_processed;
42        nonDiagSec_processed_view.setValue( true );
43
44        // Launch kernels for all sections below the diagonal section - each section has its stream
45        for( sec_start = dSec_end, sec_end = min( num_cols, dSec_end + sec_size ), sec_id = 0; sec_start < sec_end; sec_start += sec_size, sec_end = min( num_cols, sec_end + sec_size ), ++sec_id )
46        {
47          // Only compute sections that are not yet processed
48          if( ! nonDiagSec_processed_host( sec_id ) ) {
49            LSecCompute_kernel< BLOCK_SIZE ><<< lSec_blockPerGrid, threads_perBlock, 0, streams[ sec_id ] >>>( A_view, LU_view, LUnext_view, dSec_start, badEl_idx_cutoff, sec_start, sec_end, process_tol, nonDiagSec_processed_view, sec_id );
50
51            NonDiagSecAssign_kernel<<< lSec_blockPerGrid, threads_perBlock, 0, streams[ sec_id ] >>>( LU_view, LUnext_view, dSec_start, badEl_idx_cutoff, sec_start, sec_end );
52          }
53        }
54        // Wait until all sections have been computed in this iteration
55        synchronizeStreams( streams, nonDiagSecs_perRow );
56      } while( ! TNL::Algorithms::reduce( nonDiagSec_processed_view, TNL::LogicalAnd{} ) );
57
58      // Pivot the bad element
59      pivotBadElement< BLOCK_SIZE >( A_view, LU_view, piv_view, badEl_idx, num_rows, num_cols, badEl_searchStart, piv_tol );
60    } while( badEl_idx < dSec_end - 1 );
61    // ... Processing of the sections to the right of the diagonal section
62  }
```

Listing 2.8: An excerpt from the definition of the overloaded `decompose()` method for the ICM_*x*PP decomposer. The excerpt highlights how the processing of lower sections, i.e., sections below a diagonal section, is implemented. The `LSecCompute_kernel()` and `NonDiagSecAssign_kernel()` kernels compute and assign values of the lower sections, respectively. The kernels are presented separately in Listings C.3 and C.5.

The code in Listing 2.8 corresponds to the instructions from Step 6 to Step 8 in the outline presented earlier.
If there is a bad element in the diagonal section, then only elements up to and including the column containing the bad element are processed in the lower sections. Thus, as shown on Lines 33 to 35, it is necessary to redefine the number of blocks in the 1st dimension of the CUDA grid accordingly.

As depicted in Figure 2.2, the sections below a diagonal section can be processed simultaneously. To achieve this, a CUDA stream (described in Section 1.2.4) is created for each section. Furthermore,

each section has its own `processed` flag in the `nonDiagSec_processed` array. This provides control over which section is computed in each iteration. Specifically, in every iteration of the inner while-loop, all sections that are not yet processed are computed by launching a kernel using their stream. Then at the end of every iteration, `cudaStreamSynchronize()` is used in the `synchronizeStreams()` function (shown at the end of Listing C.1) to halt the CPU thread at that line until the kernels launched on the array of streams have been executed. Then, using parallel reduction, the processed states of all lower sections are evaluated. If all of the lower sections are processed, then the inner while-loop terminates and the bad element is subsequently pivoted.

The outer while loop, with the condition `badEl_idx < dSec_end - 1`, terminates if either of the following conditions is fulfilled:

1. The bad element pivoted in the current iteration is the bottom-right-most element of the diagonal section.
2. There are no bad elements in the diagonal section as `badEl_idx` is set to the illegal value, `num_cols`.

The first condition arises from the following observation: If the bad element is the bottom-right-most element of the diagonal section, then the diagonal section and the sections below it are processed once the bad element is pivoted. In other words, once the bad element is pivoted, there is no need to compute the next iteration as the values produced would be identical to the values of the current iteration.

When the diagonal and lower sections are fully processed, i.e., the diagonal section contains no bad elements, then the sections to the right of the diagonal section are processed as shown in Listing C.1. Subsequently, the implementation moves on to the next diagonal section and the process repeats itself.

Computing One Iteration of a Section While the previous part described how a matrix is processed by sections in the implementation of the ICM_*x*PP decomposer, this part describes how one iteration of a single section is computed. Specifically, this part aims to present the kernels referenced in the previous part.

As mentioned earlier, in the implementation of the ICM_*x*PP decomposer, there are three types of sections: *diagonal*, *lower*, and *right*. Each section has its own compute kernel that computes the values of the next iteration. Initially, there was one kernel that was used by all section types (see Listing 2.3 on page 65 in *Parallel LU Decomposition for the GPU* [2]), however, this approach was found to be suboptimal during the optimization process of this thesis. The implementation of the kernel contained conditions that were dependent on the type of the section. Seeing as the kernel can be executed up to thousands of times during the decomposition of a matrix, conditions in code can hinder performance. Thus, the compute kernel was split into three section-specific kernels.

One iteration of the diagonal section is computed by the `DSecCompute_kernel`, shown in Listing 2.9, using the formulas from Equations 1.11 and 1.12.

```
1 template< const int BLOCK_SIZE, typename ConstMatrixView, typename MatrixView↩
      , typename BoolArrayView, typename Index, typename Real >
2 __global__
3 void
4 DSecCompute_kernel( const ConstMatrixView A, MatrixView LU, MatrixView LUnext↩
      , const Index sec_start_col, const Index sec_end_col, const Index ↩
      sec_start_row, const Index sec_end_row, const Real process_tol, ↩
      BoolArrayView processed, const Index badEl_rowcol )
```

```
{
  Index ty = threadIdx.y;
  Index tx = threadIdx.x;

  // The IDs of each thread are set to the top-left part of its block
  Index row = blockIdx.y * blockDim.y + sec_start_row;
  Index col = blockIdx.x * blockDim.x + sec_start_col;

  // Terminate threads that are part of a block whose first element is to the bottom-right of the bad element
  // These threads would compute values that would not be saved
  if( row > badEl_rowcol && col > badEl_rowcol )
    return;

  // Each thread computes one element (row, col)
  row += ty;
  col += tx;

  // Set the IDs of threads that overreach the bounds to the closest boundary
  Index max_col = sec_end_col - 1;
  Index max_row = sec_end_row - 1;
  Index row_adj = min( row, max_row );
  Index col_adj = min( col, max_col );

  // Offset the smallest index of the thread's element by BLOCK_SIZE to allow for loop unrolling
  Index min_row_col = min( row_adj, col_adj ) - BLOCK_SIZE;

  // Declare shared memory for blocks of L and U
  __shared__ Real L_block[ BLOCK_SIZE ][ BLOCK_SIZE ];
  __shared__ Real U_block[ BLOCK_SIZE ][ BLOCK_SIZE ];

  Index i, k; Real sum = 0;

  // Compute the sum needed for the element (row, col) by loading blocks of elements from global to shared memory and multiplying them
  for( i = 0; i <= min_row_col; i += BLOCK_SIZE ) {
    L_block[ ty ][ tx ] = LU( row_adj, i + tx );
    U_block[ ty ][ tx ] = LU( i + ty, col_adj );

    __syncthreads();

    #pragma unroll( BLOCK_SIZE )
    for( k = 0; k < BLOCK_SIZE; ++k )
      sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];
    __syncthreads();
  }

  // Loops are unrolled by multiples of BLOCK_SIZE and the remaining elements are computed separately
  L_block[ ty ][ tx ] = LU( row_adj, min( i + tx, max_col ) );
  U_block[ ty ][ tx ] = LU( min( i + ty, max_row ), col_adj );

  __syncthreads();

  // Terminate threads that overreach the bounds as they have served their purpose of reading data
  if( row >= sec_end_row || col >= sec_end_col )
    return;

  // Terminate threads that reach past the bad element as they have served their purpose of reading data
```

```
61   if( row >= badEl_rowcol && col > badEl_rowcol )
62     return;
63
64   // Read LU( row, col ) from shared memory instead of global memory
65   Real LU_rowCol;
66   // Index denoting where to stop the computation of element (row, col)
67   Index t_to_use;
68
69   if( row >= col ) { // The element computed is in L
70     t_to_use = tx;
71     LU_rowCol = L_block[ ty ][ tx ];
72   } else { // The element computed is in U
73     t_to_use = ty;
74     LU_rowCol = U_block[ ty ][ tx ];
75   }
76
77   for( k = 0; k < t_to_use; ++k )
78     sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];
79
80   // The formula for L is also a part of the formula for U
81   sum = A( row, col ) - sum;
82
83   // Remaining part of the formula for U
84   if( row < col ) {
85     Real divisor = LU( row, row );
86     if( ! abs( divisor ) == 0 )
87       sum /= divisor;
88     else
89       printf( "-!> DIVISION BY ZERO\n" );
90   }
91
92   // Check if the element (row, col) has been processed
93   if( abs( LU_rowCol - sum ) > process_tol )
94     processed( 0 ) = false;
95
96   // Assign the element for the next iteration
97   LUnext( row, col ) = sum;
98 }
```

Listing 2.9: The implementation of the `DSecCompute_kernel()` kernel which computes one iteration of a diagonal section. Note that the matrices, vectors, and arrays are passed using their views, and the scalar values are copied to the local memory of each thread.

In the `decompose()` method for the ICM_*x*PP decomposer, the `BLOCK_SIZE` template parameter determines the size of the 1st and 2nd dimensions of the CUDA thread block. However, it is also used in the kernels to determine the size of the two-dimensional arrays used to temporarily store values. Furthermore, it is used to speed up execution in the form of loop unrolling as shown on Line 44 in Listing 2.9.

The core part of the kernel is the use of shared memory. As detailed at the end of Section 2.2.1 in *Parallel LU Decomposition for the GPU* [2], the computing of the sum used in the formula of each element is identical to that of matrix multiplication with the exception that not all elements in the row and column are used. In particular, each block of threads is responsible for computing a block of elements in `LUnext`. Thus, to use shared memory and avoid unnecessary accesses to global memory, the block of threads performs matrix multiplication by iterating over blocks of elements. In other words, it loads a block of elements, multiplies them, and then moves on to the next block. The value a thread computes for each block of elements is added to a local variable, `sum`. Matrix multiplication by blocks of elements using shared memory is depicted in Figure 2.3.

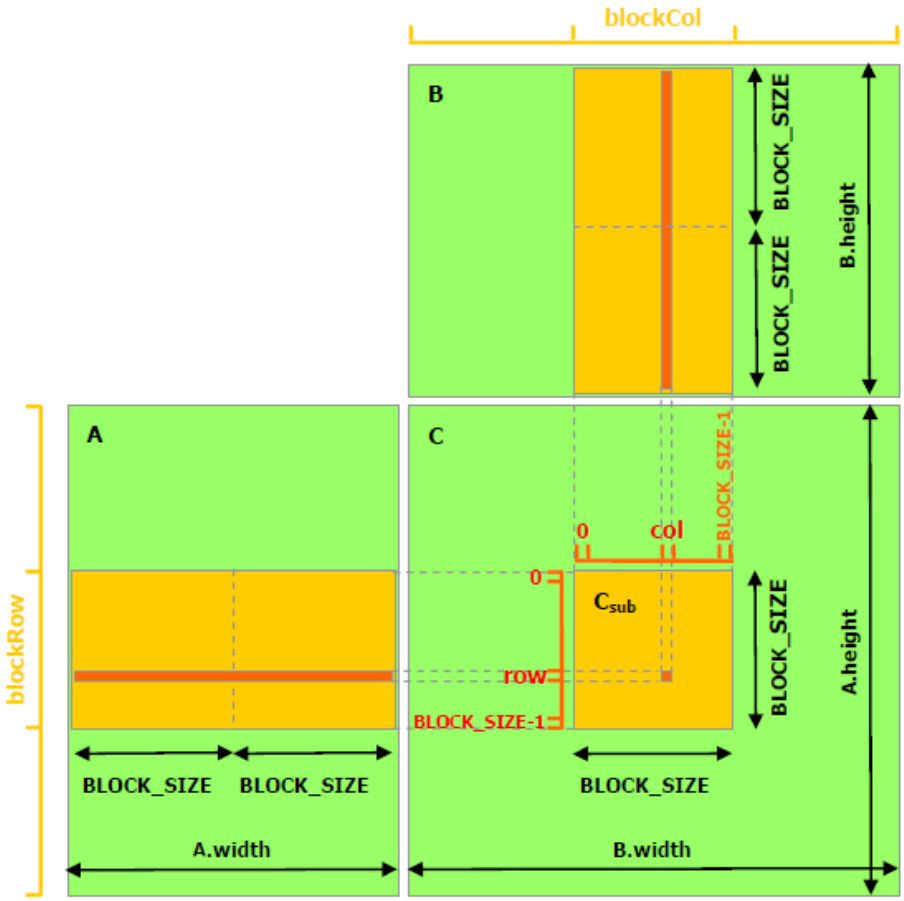

Figure 2.3: Visualization of matrix multiplication by blocks of elements using shared memory. Note that this figure, taken from *CUDA C++ Programming Guide* [3], depicts regular matrix multiplication, i.e., not the limited rendition present in Crout's method. The implementation in Listing 2.9 performs matrix multiplication until the element that is being computed in terms of LU decomposition, as shown in Figure 1.13 on page 35.

At the end of the kernel shown in Listing 2.9, it can be seen that the computed value, stored in the `sum` local variable, is saved into `LUnext` rather than `LU`. This is done on purpose to avoid inconsistent results as threads would otherwise be able to overwrite values before other threads would be able to use them to compute their elements. While it can be argued that this behavior may lead to higher performance due to fewer iterations being required, it is not stable and therefore a correct result cannot be guaranteed. Thus, the assignment of values to `LU` for the next iteration takes place in the `DSecAssign_kernel()` which is executed after the `DSecCompute_kernel()`. The `DSecAssign_kernel()` kernel is shown in Listing 2.10.

```
1 template< typename MatrixView, typename Index >
2 __global__
3 void
4 DSecAssign_kernel( MatrixView LU, MatrixView LUnext, const Index ↩
      sec_start_col, const Index sec_end_col, const Index sec_start_row, const ↩
      Index sec_end_row, const Index badEl_rowcol )
5 {
6   // Each thread computes one element (row, col)
7   Index row = blockIdx.y * blockDim.y + threadIdx.y + sec_start_row;
8   Index col = blockIdx.x * blockDim.x + threadIdx.x + sec_start_col;
9
```

```
10    // Terminate threads that overreach the bounds
11    if( row >= sec_end_row || col >= sec_end_col )
12      return;
13
14    // Terminate threads that would write elements to the bottom-right of the ↩
          bad element
15    if( row >= badEl_rowcol && col > badEl_rowcol )
16      return;
17
18    // Assign the values of the next iteration to the matrix representing the ↩
          current iteration
19    LU( row, col ) = LUnext( row, col );
20  }
```

Listing 2.10: The implementation of the `DSecAssign_kernel()` kernel that assigns values of the next iteration to the matrix representing the current iteration. Note that the matrices are passed using their views and the indices are copied to the local memory of each thread.

It is noteworthy that the values located to the bottom-right of the bad element are excluded from the assignment. The reasoning behind this is that those values are not computed by the preceding kernel as they would not be high-quality. Therefore, their assignment is omitted to mitigate unnecessary accesses to global memory.

The kernels for computing the lower and right sections are similar to that of the diagonal section, therefore, they are included in Listings C.3 and C.4, respectively. Similarly, the assignment kernel for non-diagonal sections is shown only in Listing C.5.

Solvers

The *Solvers* group consists of implementations that, given the output of an LU decomposition algorithm, solve a linear system. To solve a linear system, solvers perform the operations described in Step 2 of the two-step process mentioned in Section 1.3.1. For convenience, the operations are summarized below.

Specifically, assuming a system of $n \in \mathbb{N}$ linear equations with n unknowns and $m \in \mathbb{N}$ right-hand sides, solvers solve a linear system defined as $\mathbf{LUX} = \mathbf{PB}$, where $\mathbf{L}$ and $\mathbf{U}$ are the output of an LU decomposition algorithm, $\mathbf{X}$ is an $n \times m$ matrix of unknowns, $\mathbf{P}$ is an $n \times n$ permutation matrix, and $\mathbf{B}$ is an $n \times m$ matrix of right-hand sides. In the context of the Decomposition project, solvers perform the following steps:

1. If partial pivoting is enabled, permute the rows of matrix $\mathbf{B}$ according to matrix $\mathbf{P}$.
2. Perform forward substitution by solving $\mathbf{LY} = \mathbf{B}$.
3. Perform backward substitution by solving $\mathbf{UX} = \mathbf{Y}$.

At the end of Step 3 matrix $\mathbf{X}$ contains the values of the unknowns in the correct order. In other words, $\mathbf{X}$ can be directly applied to $\mathbf{AX} = \mathbf{B}$ without the need for permutation.

The full list of solvers and their characteristics is presented in Table 2.2.

As can be seen from Table 2.2, two solvers were implemented as part of this project: SSPP and IS_*x*PP.

Solver	With / Without PP	Unit diag. in	Device supported
SS(PP)	Yes / Yes	**U**	CPU
CuBLAStrsm(PP)	Yes / Yes	**L**, **U**	GPU
CuSolverDnXgetrs(PP)	Yes / Yes	**L**	GPU
IS_*x*(PP)	Yes / Yes	**U**	CPU, GPU

Table 2.2: The solvers made available by the Decomposition project. The solvers in **bold** font were implemented by the author of the project. The "PP" suffix indicates whether partial pivoting is used. The "*x*" in the name of certain solvers signifies their CUDA thread configuration. The "Unit diag. In" column indicates the format of the input matrix the solver supports, i.e., if the unit diagonal should be in **L**, **U**, or either.

Sequential Solver with Partial Pivoting (SSPP) Similarly to the CMPP decomposer, the Sequential Solver with partial pivoting (SSPP) is a basic algorithm implemented only for the CPU as its sequential algorithm would be inefficient on the GPU.

The purpose of the SSPP solver is to provide an accurate and simple solver. Therefore, it was not optimized or improved upon except for adding support for multiple right-hand sides. The definition of the `solve()` method for the SSPP solver is shown in Listing D.1 of Appendix D. The implementation showcases how multiple right-hand sides can be handled in forward substitution using `TNL::Algorithms::parallelFor()` (described in Section 2.1.1).

Iterative Solver with Partial Pivoting (IS_*x*PP) The IS_*x*PP solver was implemented as an experimental method that aimed to combine the approach of ICM_*x*PP and the algorithm of SSPP. However, it was not a part of the diploma thesis assignment and was therefore not prioritized in terms of optimization. Thus, the implementation of IS_*x*PP is naive in the sense that it performs iterations of both substitutions until their respective matrices are processed. In other words, it repeatedly computes all values in $\mathbf{LY} = \mathbf{B}$ simultaneously until the values of $\mathbf{Y}$ stop changing between iterations according to a set tolerance. Then, it performs the same for $\mathbf{UX} = \mathbf{Y}$ and $\mathbf{X}$. While the solver is implemented both for the CPU and the GPU, the former implementation serves solely as a means to verify the approach, therefore, it is not presented in this thesis. The definition of the `solve()` method for the IS_*x*PP solver is shown in Listing E.1.

Compatibility of Decomposers and Solvers Combining the information from Tables 2.1 and 2.2 yields the overview of compatible decomposers and solvers presented in Table 2.3.

Decomposer \ Solver	SS(PP)	IS_*x*(PP)	CuBLAStrsm(PP)	CuSolverDnXgetrs(PP)
CM(PP)	Yes	Yes	Yes	No
PCM_*x*(PP)	Yes	Yes	Yes	No
ICM_*x*(PP)	Yes	Yes	Yes	No
CuSolverDnXgetrf(PP)	No	No	Yes	Yes

Table 2.3: Compatibility of decomposer and solvers made available by the Decomposition project. To clarify, a decomposer-solver combination is compatible if the two can be used to complete a linear solver.

2.2.3 Benchmarks

One of the tasks for this thesis was to compare the performance of decomposers and solvers that were implemented by the author with that of established libraries. For this purpose, a benchmark system was implemented for both decomposers and solvers. It is important to note that while the benchmark system was implemented as part of *Parallel LU Decomposition for the GPU* [2], it was adapted from the SpMV TNL Benchmark structure[14]. However, with the introduction of partial pivoting into the project, the implementation underwent major refactoring.

This section briefly introduces the benchmark procedure for both decomposers and solvers. First, steps that are common for both benchmarks are described. Then, aspects specific to each benchmark are mentioned separately.

Both benchmarks are initiated via a bash script that sets up the logging directory, compiles the benchmarks, and, finally, launches the benchmark for each matrix found in a set directory. Note that the benchmark is launched twice: first using double precision, then using single precision. Each execution of the benchmark for a given matrix consists of the following steps:

1. Load the matrix from its `.mtx` file. The `mtx` file format is referred to as the *Matrix Market File Format*[15] and it is primarily used to represent sparse matrices as only nonzero entries are stored along with their coordinates.
2. Perform the benchmarked operation using a baseline implementation, for example, CMPP.
3. Perform the benchmarked operation using a selected implementation.

Each benchmarked operation can be performed in loops with the results of all loops averaged to amount for statistically significant results. The results can include, for example, the execution time of the operation and its standard deviation, the maximum difference between the actual results and expected results, etc. The averaged results are logged using `TNL::Benchmarks::JsonLogging` as JSON objects into log files which can be later transformed into HTML tables and MATLAB plots using a Python script provided in `src/Benchmarks/utils`.

Note that the maximum difference is computed on the CPU to assure the highest possible accuracy. The maximum difference is defined for decomposers as:

$$\max |\mathbf{A} - \mathbf{PLU}| \ , \tag{2.6}$$

where $\mathbf{A}$ is the input matrix, $\mathbf{P}$ is the permutation matrix (represented in code by a pivoting vector), and matrices $\mathbf{L}$ and $\mathbf{U}$ are the results of the LU decomposition operation. To clarify, the maximum difference for decomposers is the largest absolute difference between a computed element and its expected result.

For solvers the maximum difference is defined as:

$$\max |\mathbf{PLUX} - \mathbf{B}| \ , \tag{2.7}$$

where matrices $\mathbf{P}$, $\mathbf{L}$, and $\mathbf{U}$ are the same as in Equation 2.6, $\mathbf{X}$ is the computed matrix of unknowns, and $\mathbf{B}$ is the matrix of right-hand sides. To clarify, the computed unknowns are used in the system of

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

[15] Matrix Market Exchange Formats URL: `https://math.nist.gov/MatrixMarket/formats.html#MMformat`

equations in matrix form, and then the largest absolute difference between the left- and right-hand sides is used.

As mentioned in *Parallel LU Decomposition for the GPU* [2], many aspects of the implementations can only be verified on larger matrices. Thus, aside from unit tests, benchmarks were a crucial part of assuring the quality of the implementations.

2.3 BDDCML Integration

Another key task for this thesis was to apply the decomposers and solvers made available by the Decomposition project in the Multilevel BDDC Solver Library (BDDCML) [47, 48, 49, 50]. According to the project's website[16], the goal of BDDCML is to provide a scalable parallel solver of linear equations for systems originating from finite element computations. The implementation of the library was inspired by the Adaptive-Multilevel variant of the Balancing Domain Decomposition by Constraints (BDDC) method. More information regarding BDDCML can be found on its website[16] or in its GitHub repository[17].

Integration with the Decomposition Project BDDCML is written in Fortran 95 and the Decomposition project is written in C++. Therefore, to use decomposers and solvers from the Decomposition project in BDDCML, the TNL-BDDCML Interface[18] was used. The interface was implemented by Ing. Jakub Šístek, Ph.D. as a means to use data structures and functionalities provided by TNL in BDDCML, for example, the allocation and deallocation of a `TNL::Matrices::DenseMatrix` instance on the GPU, GEMV, etc.
For the Decomposition project, Fortran wrappers calling the `decompose()` method of a specific decomposer or the `solve()` method of a specific solver were added to the interface.

Note that both the interface and BDDCML needed to be extended. Therefore, the repositories were forked and the changes related to the Decomposition project were added separately. The final versions of both repositories that contain changes related to the Decomposition project are available on request or as an attachment to this thesis.

Benchmarks The performance of decomposers and solvers provided by the Decomposition project is compared with the corresponding routines of the MAGMA library [51] which was already used in BDDCML. In particular, the performance was compared when solving the "Poisson equation on a cube" problem present in BDDCML. Furthermore, for the results of the performance comparison to be statistically significant, a benchmarking system was implemented. The system comprises:

- `prepare_poisson_on_cube_benchmark` - The Bash script that compiles BDDCML with different combinations of decomposers and solvers.
- `run_poisson_on_cube_benchmark` - The Bash script that runs the `poisson_on_cube` executable in loops and sorts logs into directories.

[16] BDDCML webpage URL: `https://users.math.cas.cz/~sistek/software/bddcml.html`
[17] Multilevel BDDC Solver Library GitHub repository URL: `https://github.com/sistek/bddcml`
[18] tnl_bddcml_interface Bitbucket repository URL: `https://bitbucket.org/tnl-decomposition/tnl_bddcml_interface/src/master`

- `poisson_on_cube_logs_to_csv.py` - The Python script that parses the logs for results, for example, total time taken by a procedure, total time of execution, the accuracy of the solution, etc. These results, obtained across several loops, are then averaged and saved into a CSV file and plotted using MATLAB.

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] CROVELLA, R. *Concurrent kernel execution without stream* [online]. [visited on 2023-06-15]. Available from: https://forums.developer.nvidia.com/t/concurrent-kernel-execution-without-stream/46879/2.

[25] CROVELLA, R. *GPU sharing among different application with different CUDA context* [online]. [visited on 2023-06-15]. Available from: https://forums.developer.nvidia.com/t/gpu-sharing-among-different-application-with-different-cuda-context/53057/4.

[26] 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.

[27] 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.

[28] 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.

[29] *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.

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

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

[32] 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.

[33] 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.

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

[35] 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.

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

[37] 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.

[38] 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.

[39] OBERHUBER, T.; KLINKOVSKÝ, J.; FUČÍK, R. TNL: NUMERICAL LIBRARY FOR MODERN PARALLEL ARCHITECTURES. *Acta Polytechnica* [online]. 2021, vol. 61, no. SI, pp. 122–134 [visited on 2023-06-06]. ISSN 1805-2363. Available from DOI: 10.14311/AP.2021.61.0122.

[40] KLINKOVSKÝ, J.; OBERHUBER, T.; FUČÍK, R.; ŽABKA, V. Configurable Open-source Data Structure for Distributed Conforming Unstructured Homogeneous Meshes with GPU Support. *ACM Transactions on Mathematical Software* [online]. 2022, vol. 48, no. 3, pp. 1–30 [visited on 2023-06-06]. ISSN 0098-3500. Available from DOI: 10.1145/3536164.

[41] *TNL aims to be STL for HPC* [online]. Template Numerical Library [visited on 2023-05-21]. Available from: https://tnl-project.org.

[42] *Template Numerical Library User Guide* [online]. 2023. [visited on 2023-05-21]. Available from: https://tnl-project.gitlab.io/tnl/index.html.

[43] *CuSOLVER API Reference* [online]. Nvidia, 2023 [visited on 2023-05-26]. Available from: https://docs.nvidia.com/cuda/archive/12.0.0/cusolver/index.html.

[44] *CuBLAS* [online]. Nvidia, 2022 [visited on 2023-05-26]. Available from: 20%20December%202022.

[45] NICELY, M.; KHADATARE, M.; SEGAL, A.; SZUPPE, J. *CUDA Library Samples* [online]. 2023. [visited on 2023-05-28]. Available from: https://github.com/NVIDIA/CUDALibrarySamples.

[46] COX, H.; OBERPARLEITER, P. *LTP GCOV extension (LCOV)* [online]. 2023. [visited on 2023-05-30]. Available from: https://github.com/linux-test-project/lcov.

[47] ŠÍSTEK, J.; SOUSEDÍK, B.; BURDA, P.; MANDEL, J.; NOVOTNÝ, J. Application of the parallel BDDC preconditioner to the Stokes flow. *Computers & Fluids* [online]. 2011, vol. 46, no. 1, pp. 429–435 [visited on 2023-06-06]. ISSN 00457930. Available from DOI: 10.1016/j.compfluid.2011.01.002.

[48] ŠÍSTEK, J.; ČERTÍKOVÁ, M.; BURDA, P.; NOVOTNÝ, J. Face-based selection of corners in 3D substructuring. *Mathematics and Computers in Simulation* [online]. 2012, vol. 82, no. 10, pp. 1799–1811 [visited on 2023-06-06]. ISSN 03784754. Available from DOI: 10.1016/j.matcom.2011.06.007.

[49] SOUSEDÍK, B.; ŠÍSTEK, J.; MANDEL, J. Adaptive-Multilevel BDDC and its parallel implementation. *Computing* [online]. 2013, vol. 95, no. 12, pp. 1087–1119 [visited on 2023-06-06]. ISSN 0010-485X. Available from DOI: 10.1007/s00607-013-0293-5.

[50] ŠÍSTEK, J.; BŘEZINA, J.; SOUSEDÍK, B. BDDC for mixed-hybrid formulation of flow in porous media with combined mesh dimensions. *Numerical Linear Algebra with Applications* [online]. 2015, vol. 22, no. 6, pp. 903–929 [visited on 2023-06-06]. ISSN 10705325. Available from DOI: 10.1002/nla.1991.

[51] TOMOV, S.; NATH, R.; LTAIEF, H.; DONGARRA, J. Dense linear algebra solvers for multicore with GPU accelerators. In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)* [online]. IEEE, 2010, pp. 1–8 [visited on 2023-06-06]. ISBN 978-1-4244-6533-0. Available from DOI: 10.1109/IPDPSW.2010.5470941.

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

[53] 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.

[54] 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.

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

[56] 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

CMPP Implementation

The implementation of CMPP and the functions it uses are shown in Listing A.1.

```
template< typename Matrix, typename Vector >
void
CroutMethod::decompose( Matrix& LU, Vector& piv )
{
  using Real = typename Matrix::RealType;
  using Index = typename Matrix::IndexType;

  const Index& num_rows = LU.getRows();
  const Index& num_cols = LU.getColumns();

  // Pivoting tolerance
  const Real piv_tol = 1.0e-5;

  auto LU_view = LU.getView();
  auto piv_view = piv.getView();

  // Fill the pivoting vector with increments of 1 starting from 1 to num_rows.
  BaseDecomposer::setDefaultPivotingValues( piv );

  Index i, j, k, sum;

  for( j = 0; j < num_cols; ++j ) {
    for( i = j; i < num_rows; ++i ) {
      sum = 0;
      for( k = 0; k < j; ++k )
        sum = sum + LU_view( i, k ) * LU_view( k, j );

      LU_view( i, j ) = LU_view( i, j ) - sum;
    }

    if( TNL::abs( LU_view( j, j ) ) <= piv_tol ) {
      pivotRowOfMatrix_host( j, LU_view, num_rows, num_cols, piv_view );

      if( LU_view( j, j ) == 0 ) {
        // Last element in the matrix does not require pivoting as no elements are computed using it, i.e., division by zero cannot occur
        if( j != num_rows - 1 )
          throw Exceptions::MatrixSingular( "LU", j, j );
      }
    }
```

```
40
41      for( i = j + 1; i < num_rows; ++i ) {
42        sum = 0;
43        for( k = 0; k < j; ++k )
44          sum = sum + LU_view( j, k ) * LU_view( k, i );
45
46        LU_view( j, i ) = ( LU_view( j, i ) - sum ) / LU_view( j, j );
47      }
48    }
49  }
50
51  template< typename Index, typename MatrixView, typename VectorView, typename ↩
        Real = typename MatrixView::RealType >
52  std::pair< Real, Index >
53  pivotRowOfMatrix_host( const Index& j, MatrixView& M, const Index& num_rows, ↩
        const Index& num_cols, VectorView& piv )
54  {
55    // Find the row below the j-th row with max. value in the j-th column
56    Real max = TNL::abs( M( j, j ) );  // or M(i,j)
57    Index piv_row = j;
58
59    for( Index i = j + 1; i < num_rows; ++i ) {
60      Real zij = TNL::abs( M( i, j ) );
61      if( zij > max ) {
62        max = zij;
63        piv_row = i;
64      }
65    }
66
67    if( piv_row != j ) {  // exchange rows j, piv_row
68      swapRows_host( M, j, piv_row, num_cols );
69      piv( j ) = piv_row + 1;
70    }
71
72    return std::make_pair( max, piv_row );
73  }
```

Listing A.1: Excerpt from the implementation of CMPP. The `pivotRowOfMatrix_host()` function, presented below the `decompose()` method, is implemented in the parent class of `Crout Method`: `BaseDecomposer`. Note that the code has been slightly modified for brevity. For example, the `swapRows_host()` function has been omitted as it is a basic operation, and the checks for appropriate sizing of matrices/vectors have been removed.

Appendix B

PCM*x*PP Implementation

The implementation of PCM*x*PP is shown in Listing B.1. Note that the kernels called in Listing B.1 are shown separately in Listings B.2 and B.3.

```
template< const int BLOCK_SIZE >
template< typename Matrix, typename VecIndex, typename Device >
void
CroutMethod< BLOCK_SIZE >::decompose( Matrix& LU, Containers::RowOrderVector<VecIndex, Device >& piv )
{
  using Real = typename Matrix::RealType;
  using Index = typename Matrix::IndexType;

  const Index& num_rows = LU.getRows();
  const Index& num_cols = LU.getColumns();

  auto LU_view = LU.getView();
  auto piv_view = piv.getView();

  // Fill the pivoting vector with increments of 1 starting from 1 to num_rows.
  BaseDecomposer::setDefaultPivotingValues( piv );

  const Real piv_tol = 1.0e-5;

  constexpr int threads_perBlock = BLOCK_SIZE * BLOCK_SIZE;
  // Round to nearest higher multiple of threads_perBlock - only if threads_perBlock is a multiple of 2
  const Index num_cols_rounded = ( num_cols + threads_perBlock - 1 ) & -threads_perBlock;
  const Index blocks_perGrid = num_cols_rounded / threads_perBlock;

  for( Index diag = 0; diag < num_rows; ++diag ) {
    ColCompute_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, diag, num_rows );

    if( TNL::abs( LU_view.getElement( diag, diag ) ) <= piv_tol ) {
      std::pair< Real, Index > max_elem = pivotRowOfMatrix_device< threads_perBlock >( diag, LU_view, num_rows, num_cols, piv_view );

      if( max_elem.first == 0 ) {
        // Last element does not require pivoting as no elements are computed using it - division by zero cannot occur for the last element
        if( diag != num_rows - 1 )
          throw Exceptions::MatrixSingular( "LU", diag, diag );
```

```
35        }
36     }
37
38     RowCompute_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, diag,
           num_cols );
39   }
40 }
41
42 template< const int THREADS_PER_BLOCK,
43       typename Index,
44       typename MatrixView,
45       typename VectorView,
46       typename Real = typename MatrixView::RealType >
47 std::pair< Real, Index >
48 pivotRowOfMatrix_device( const Index& j, MatrixView& M, const Index& num_rows
      , const Index& num_cols, VectorView& piv )
49 {
50   auto fetchAbsElement = [ = ] __cuda_callable__( Index row )
51   {
52     return abs( M( row, j ) );
53   };
54   std::pair< Real, Index > max_elem =
55     TNL::Algorithms::reduceWithArgument< TNL::Devices::Cuda >( j, num_rows,
          fetchAbsElement, TNL::MaxWithArg{} );
56
57   // Only need to swap rows if the pivoting row is different from j
58   if( max_elem.second != j ) {
59     swapRows_device< THREADS_PER_BLOCK >( M, j, max_elem.second, num_cols );
60     piv( j ) = max_elem.second + 1;
61   }
62
63   return max_elem;
64 }
```

Listing B.1: Excerpt from the implementation of PCMxPP. The template parameter `BLOCK_SIZE` is equivalent to x in PCMxPP. On input, matrix `LU` is assumed to contain the values of $\mathbf{A}$, and `piv` is expected to be appropriately sized. Furthermore, unlike `LU`, `piv` is assumed to be allocated on the Host. The `pivotRowOfMatrix_device()` function, presented below the `decompose()` method, is implemented in the parent class of `CroutMethod`: `BaseDecomposer`. Note that the code has been slightly modified for brevity. For example, the `swapRows_device()` function has been omitted as it is a basic operation, and the checks for appropriate sizing of matrices/vectors have been removed.

```
1  template< typename MatrixView, typename Index >
2  __global__
3  void
4  ColCompute_kernel( MatrixView LU, const Index col, const Index num_rows )
5  {
6    using Real = typename MatrixView::RealType;
7    // Offset threads to start from the diagonal element (including it)
8    const Index row = blockIdx.x * blockDim.x + threadIdx.x + col;
9
10   if( row >= num_rows )
11     return;
12
13   Real sum = 0;
14   for( Index k = 0; k < col; ++k )
15     sum += LU( row, k ) * LU( k, col );
16
17   LU( row, col ) = LU( row, col ) - sum;
```

```
}
```

Listing B.2: The implementation of the `ColCompute_kernel()` kernel which computes one column of `LU`.

```
template< typename MatrixView, typename Index >
__global__
void
RowCompute_kernel( MatrixView LU, Index row, const Index num_cols )
{
  using Real = typename MatrixView::RealType;
  // Offset threads to start from element to the right of the diagonal element
  const Index col = blockIdx.x * blockDim.x + threadIdx.x + row + 1;

  if( col >= num_cols )
    return;

  Real sum = 0;
  for( Index k = 0; k < row; ++k )
    sum += LU( row, k ) * LU( k, col );

  LU( row, col ) = ( LU( row, col ) - sum ) / LU( row, row );
}
```

Listing B.3: The implementation of the `RowCompute_kernel()` kernel which computes one row of `LU`.

Appendix C

ICM*x*PP Implementation

The implementation of ICM*x*PP is shown in Listing C.1. Note that the kernels called in Listing C.1 are shown separately in Listings 2.9, 2.10, C.3, and C.4.

```
template< const int BLOCK_SIZE >
template< typename Matrix, typename Vector >
void
IterativeCroutMethod< BLOCK_SIZE >::decompose( Matrix& A, Matrix& LU, Vector&↩
    piv )
{
  using Real = typename Matrix::RealType;
  using Index = typename Matrix::IndexType;

  const Index num_rows = LU.getRows();
  const Index num_cols = LU.getColumns();

  // Matrix representing the next iteration
  Matrix LUnext;
  LUnext.setLike( LU );

  // Processing tolerance
  const Real process_tol = 0.0;
  // Pivoting tolerance
  const Real piv_tol = 1.0e-5;

  // Flag to indicate that the diagonal section has been processed
  TNL::Containers::Array< bool, TNL::Devices::Cuda > processed{ 1, false };

  // Fill the pivoting vector with increments of 1 starting from 1 to ↩
      num_rows.
  BaseDecomposer::setDefaultPivotingValues( piv );

  // Determine the size of the section based on the dimensions of the matrix
  Index sec_size = min( max( num_cols / 10, (Index) 256 ), (Index) 1024 );
  sec_size = ( sec_size + BLOCK_SIZE - 1 ) / BLOCK_SIZE * BLOCK_SIZE;

  // CUDA grid configuration
  Index blocks = sec_size / BLOCK_SIZE;
  dim3 threads_perBlock( BLOCK_SIZE, BLOCK_SIZE );
  dim3 blocks_perGrid( blocks, blocks );

  // Allocate and initialize an array of stream handles to process sections ↩
      in parallel
  const Index nonDiagSecs_perRow = TNL::ceil( (double) num_cols / (double) ↩
      sec_size ) - 1;
```

```
38  auto* streams = (cudaStream_t*) malloc( nonDiagSecs_perRow * sizeof( ↩
        cudaStream_t ) );
39  for( Index i = 0; i < nonDiagSecs_perRow; ++i )
40    cudaStreamCreate( &( streams[ i ] ) );
41
42  // Flags indicating the processing status of non-diagonal sections
43  TNL::Containers::Array< bool, TNL::Devices::Cuda, Index > ↩
        nonDiagSec_processed{ nonDiagSecs_perRow, false };
44  TNL::Containers::Array< bool, TNL::Devices::Host, Index > ↩
        nonDiagSec_processed_host{ nonDiagSecs_perRow, false };
45
46  // Views are used from here on
47  auto A_view = A.getView();
48  auto LU_view = LU.getView();
49  auto LUnext_view = LUnext.getView();
50  auto piv_view = piv.getView();
51  auto processed_view = processed.getView();
52  auto nonDiagSec_processed_view = nonDiagSec_processed.getView();
53
54  // Lambda for fetching the index of bad elements
55  auto get_badEl_colIdxs = [ = ] __cuda_callable__( Index col ) -> Index
56  {
57    return abs( LU_view( col, col ) ) <= piv_tol ? col : num_cols;
58  };
59
60  // Diagonal section start and end
61  Index dSec_start, dSec_end;
62  // Non-diagonal section start, end, and id
63  Index sec_start, sec_end, sec_id;
64  Index badEl_idx;
65
66  // Loop through the diagonal sections
67  for( dSec_start = 0, dSec_end = min( num_cols, sec_size ); dSec_start < ↩
        dSec_end; dSec_start += sec_size, dSec_end = min( num_cols, dSec_end + ↩
        sec_size ) )
68  {
69    // Set the starting point of the search for the first bad element
70    Index badEl_searchStart = dSec_start;
71
72    do { // Process the diagonal section and the sections below it
73      // Get next badEl_idx
74      badEl_idx = TNL::Algorithms::reduce< TNL::Devices::Cuda >( ↩
            badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, ↩
            num_cols );
75
76      do {  // Process the diagonal section
77        // Reset the processed flag
78        processed_view.setElement( 0, true );
79
80        // Compute the values up to and including the bad element
81        DSecCompute_kernel< BLOCK_SIZE ><<< blocks_perGrid, threads_perBlock ↩
              >>>( A_view, LU_view, LUnext_view, dSec_start, dSec_end, ↩
              dSec_start, dSec_end, process_tol, processed_view, badEl_idx );
82
83        // Assign the values computed for the next iteration
84        DSecAssign_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, ↩
              LUnext_view, dSec_start, dSec_end, dSec_start, dSec_end, ↩
              badEl_idx );
85
86        // Check if a bad element is present between the previous bad element↩
               and the current bad element
```

```
87          Index badEl_idx_new = TNL::Algorithms::reduce< TNL::Devices::Cuda >( ↩
                badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, ↩
                num_cols );
88          // If a bad element was detected before the current one -> set it as ↩
                the new bad element
89          if( badEl_idx_new < badEl_idx )
90            badEl_idx = badEl_idx_new;
91        } while( ! processed_view.getElement( 0 ) );
92
93        // The Diagonal section contains processed values excluding the values ↩
              to the bottom-right of the bad element
94        // Compute the lower sections up to and including the column containing↩
               the bad element
95
96        // Limit the number of threads used based on the number of columns that↩
               will be computed
97        Index badEl_idx_cutoff = min( badEl_idx + 1, dSec_end );
98        Index lSec_width_rounded = ( badEl_idx_cutoff - dSec_start + BLOCK_SIZE↩
              - 1 ) & -BLOCK_SIZE;
99        dim3 lSec_blockPerGrid( TNL::max( lSec_width_rounded / BLOCK_SIZE, (↩
              Index) 1 ), blocks );
100
101       // Default to false so that all kernels are run in the first iteration
102       nonDiagSec_processed_view.setValue( false );
103
104       do { // Process the lower sections in parallel
105         nonDiagSec_processed_host = nonDiagSec_processed;
106         nonDiagSec_processed_view.setValue( true );
107
108         for( sec_start = dSec_end, sec_end = min( num_cols, dSec_end + ↩
                sec_size ), sec_id = 0; sec_start < sec_end; sec_start += ↩
                sec_size, sec_end = min( num_cols, sec_end + sec_size ), ++sec_id↩
                 )
109         {
110           // Only compute sections that are not yet processed
111           if( ! nonDiagSec_processed_host( sec_id ) ) {
112             LSecCompute_kernel< BLOCK_SIZE ><<< lSec_blockPerGrid, ↩
                    threads_perBlock, 0, streams[ sec_id ] >>>( A_view, LU_view, ↩
                    LUnext_view, dSec_start, badEl_idx_cutoff, sec_start, sec_end↩
                    , process_tol, nonDiagSec_processed_view, sec_id );
113
114             NonDiagSecAssign_kernel<<< lSec_blockPerGrid, threads_perBlock, ↩
                    0, streams[ sec_id ] >>>( LU_view, LUnext_view, dSec_start, ↩
                    badEl_idx_cutoff, sec_start, sec_end );
115           }
116         }
117         // Wait until all sections have been computed in this iteration
118         synchronizeStreams( streams, nonDiagSecs_perRow );
119       } while( ! TNL::Algorithms::reduce( nonDiagSec_processed_view, TNL::↩
              LogicalAnd{} ) );
120
121       // Pivot the bad element
122       pivotBadElement< BLOCK_SIZE >( A_view, LU_view, piv_view, badEl_idx, ↩
              num_rows, num_cols, badEl_searchStart, piv_tol );
123
124     } while( badEl_idx < dSec_end - 1 );
125
126     nonDiagSec_processed_view.setValue( false );
127
128     // Process sections to the right of the diagonal section in parallel
129     // At this point there are no bad elements in the diagonal section
130     do {
```

```
131       nonDiagSec_processed_host = nonDiagSec_processed;
132       nonDiagSec_processed_view.setValue( true );
133
134       // Launch kernels for all sections below the diagonal section - each ↩
              section has its stream
135       for( sec_start = dSec_end, sec_end = min( num_cols, dSec_end + sec_size↩
              ), sec_id = 0; sec_start < sec_end; sec_start += sec_size, sec_end↩
              = min( num_cols, sec_end + sec_size ), ++sec_id )
136       {
137         // Only compute sections that are not yet processed
138         if( ! nonDiagSec_processed_host( sec_id ) ) {
139           RSecCompute_kernel< BLOCK_SIZE ><<< blocks_perGrid, ↩
                  threads_perBlock, 0, streams[ sec_id ] >>>( A_view, LU_view, ↩
                  LUnext_view, sec_start, sec_end, dSec_start, dSec_end, ↩
                  process_tol, nonDiagSec_processed_view, sec_id );
140
141           NonDiagSecAssign_kernel<<< blocks_perGrid, threads_perBlock, 0, ↩
                  streams[ sec_id ] >>>( LU_view, LUnext_view, sec_start, sec_end↩
                  , dSec_start, dSec_end );
142         }
143       }
144       // Wait until all sections have been computed in this iteration
145       synchronizeStreams( streams, nonDiagSecs_perRow );
146     } while( ! TNL::Algorithms::reduce( nonDiagSec_processed_view, TNL::↩
          LogicalAnd{} ) );
147   }
148
149   // Release resources
150   for( Index i = 0; i < nonDiagSecs_perRow; ++i )
151     cudaStreamDestroy( streams[ i ] );
152
153   free( streams );
154 }
155
156 void
157 synchronizeStreams( cudaStream_t* streams, const int num_streams )
158 {
159   for( int i = 0; i < num_streams; i++ )
160     cudaStreamSynchronize( streams[ i ] );
161 }
```

Listing C.1: Excerpt from the implementation of ICMxPP. The template parameter `BLOCK_SIZE` is equivalent to x in ICMxPP. On input, matrix `A` is assumed to contain the values of $\mathbf{A}$, matrix `LU` is assumed to contain the initial estimate of the decomposition, and `piv` is expected to be appropriately sized and allocated on the Host. On output, matrix `LU` contains the values of matrices $\mathbf{L}$ and $\mathbf{U}$ in the format presented in Equation 2.4, and `piv` contains the row permutations. The `synchronizeStreams()` function is included below the `decompose()` method. The `pivotBadElement()` function is shown in Listing C.2. The code has been slightly modified for brevity, for example, the checks for appropriate sizing of matrices and vectors have been removed.

```
1 template< const int BLOCK_SIZE, typename MatrixView, typename VectorView, ↩
      typename Index, typename Real >
2 void
3 pivotBadElement( MatrixView& A, MatrixView& LU, VectorView& piv, const Index&↩
      j, const Index& num_rows, const Index& num_cols, Index& ↩
      badEl_searchStart, const Real& piv_tol )
4 {
5   // The last element of the matrix does not require pivoting as no elements ↩
        are computed using it, i.e., division by zero cannot occur for the last↩
        element
```

```
  if( j >= num_cols - 1 ) {
    return;
  }

  std::pair< Real, Index > max_elem = pivotRowOfMatrices_device< BLOCK_SIZE *↩
      BLOCK_SIZE >( j, A, LU, num_rows, num_cols, piv );

  if( max_elem.first == 0 ) {
    // The current bad element in the main diagonal is zero even after ↩
        pivoting -> Decomposition failed
    throw Exceptions::MatrixSingular( "LU", j, j );
  }
  else if( max_elem.first <= piv_tol ) {
    // The value at the index of the bad element is smaller than the pivoting↩
        tolerance, however, it is not zero, so the computation can continue
    // Search for the next bad element after this one
    badEl_searchStart = j + 1;
  }
}

template< const int THREADS_PER_BLOCK, typename Index, typename MatrixView, ↩
    typename VectorView, typename Real = typename MatrixView::RealType >
std::pair< Real, Index >
pivotRowOfMatrices_device( const Index& j, MatrixView& A, MatrixView& M, ↩
    const Index& num_rows, const Index& num_cols, VectorView& piv )
{
  auto fetchAbsElement = [ = ] __cuda_callable__( Index row )
  {
    return abs( M( row, j ) );
  };

  std::pair< Real, Index > max_elem = TNL::Algorithms::reduceWithArgument< ↩
      TNL::Devices::Cuda >( j, num_rows, fetchAbsElement, TNL::MaxWithArg{} )↩
      ;

  // Only need to swap rows if the pivoting row is different from j
  if( max_elem.second != j ) {
    swapRows_device< THREADS_PER_BLOCK >( A, M, j, max_elem.second, num_cols ↩
        );
    piv( j ) = max_elem.second + 1;
  }

  return max_elem;
}
```

Listing C.2: The definition of the `pivotBadElement()` function which is responsible for pivoting a bad element found in column `j` of the main diagonal. The `pivotRowOfMatrices_device()` function, presented below the `pivotBadElement()` function, is implemented in the parent class of `IterativeCroutMethod`: `BaseDecomposer`. Note that the variant of the `swapRows_device()` function presented in the `pivotRowOfMatrices_device()` swaps the rows in two matrices.

```
template< const int BLOCK_SIZE, typename ConstMatrixView, typename MatrixView↩
    , typename BoolArrayView, typename Index, typename Real >
__global__
void
LSecCompute_kernel( const ConstMatrixView A, MatrixView LU, MatrixView LUnext↩
    , const Index sec_start_col, const Index sec_end_col, const Index ↩
    sec_start_row, const Index sec_end_row, const Real process_tol, ↩
    BoolArrayView processed, const Index sec_id )
{
  Index ty = threadIdx.y;
```

```
7   Index tx = threadIdx.x;
8
9   // Each thread computes one element (row, col)
10  Index row = blockIdx.y * blockDim.y + ty + sec_start_row;
11  Index col = blockIdx.x * blockDim.x + tx + sec_start_col;
12
13  // Set the IDs of threads that overreach the bounds to the closest boundary
14  Index max_col = sec_end_col - 1;
15  Index max_row = sec_end_row - 1;
16  Index row_adj = min( row, max_row );
17  Index col_adj = min( col, max_col );
18
19  // Offset the smallest index of the thread's element by BLOCK_SIZE to allow↩
        for loop unrolling
20  // In a lower section, the column index is always smaller
21  Index min_row_col = col_adj - BLOCK_SIZE;
22
23  __shared__ Real L_block[ BLOCK_SIZE ][ BLOCK_SIZE ];
24  __shared__ Real U_block[ BLOCK_SIZE ][ BLOCK_SIZE ];
25
26  Index i, k; Real sum = 0;
27
28  // Compute the sum needed for the element (row, col) by loading blocks of ↩
        elements from global to shared memory and multiplying them
29  for( i = 0; i <= min_row_col; i += BLOCK_SIZE ) {
30    L_block[ ty ][ tx ] = LU( row_adj, i + tx );
31    U_block[ ty ][ tx ] = LU( i + ty, col_adj );
32
33    __syncthreads();
34
35    #pragma unroll( BLOCK_SIZE )
36    for( k = 0; k < BLOCK_SIZE; ++k )
37      sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];
38    __syncthreads();
39  }
40
41  // Loops are unrolled by multiples of BLOCK_SIZE and the remaining elements↩
        are computed separately
42  L_block[ ty ][ tx ] = LU( row_adj, min( i + tx, max_col ) );
43  U_block[ ty ][ tx ] = LU( min( i + ty, max_row ), col_adj );
44
45  __syncthreads();
46
47  // Terminate threads that overreach the bounds as they have served their ↩
        purpose of reading data
48  if( row >= sec_end_row || col >= sec_end_col )
49    return;
50
51  for( k = 0; k < tx; ++k )
52    sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];
53
54  // Formula for L
55  sum = A( row, col ) - sum;
56
57  // Check if the element (row, col) has been processed
58  // Read LU( row, col ) from shared memory instead of global memory
59  if( abs( L_block[ ty ][ tx ] - sum ) > process_tol )
60    processed( sec_id ) = false;
61
62  // Assign the element for the next iteration
63  LUnext( row, col ) = sum;
64 }
```

Listing C.3: The implementation of the `LSecCompute_kernel()` kernel which computes one iteration of a lower section. Note that the matrices, vectors, and arrays are passed using their views, and the scalar values are copied to the local memory of each thread.

```
template< const int BLOCK_SIZE, typename ConstMatrixView, typename MatrixView, typename BoolArrayView, typename Index, typename Real >
__global__
void
RSecCompute_kernel( const ConstMatrixView A, MatrixView LU, MatrixView LUnext, const Index sec_start_col, const Index sec_end_col, const Index sec_start_row, const Index sec_end_row, const Real process_tol, BoolArrayView processed, const Index sec_id )
{
  Index tx = threadIdx.x;
  Index ty = threadIdx.y;

  // Each thread computes one element (row, col)
  Index row = blockIdx.y * blockDim.y + ty + sec_start_row;
  Index col = blockIdx.x * blockDim.x + tx + sec_start_col;

  // Set the IDs of threads that overreach the bounds to the closest boundary
  Index max_col = sec_end_col - 1;
  Index max_row = sec_end_row - 1;
  Index row_adj = min( row, max_row );
  Index col_adj = min( col, max_col );

  // Offset the smallest index of the thread's element by BLOCK_SIZE to allow for loop unrolling
  // In a right section, the row index is always smaller
  Index min_row_col = row_adj - BLOCK_SIZE;

  __shared__ Real L_block[ BLOCK_SIZE ][ BLOCK_SIZE ];
  __shared__ Real U_block[ BLOCK_SIZE ][ BLOCK_SIZE ];

  Index i, k; Real sum = 0;

  // Compute the sum needed for the element (row, col) by loading blocks of elements from global to shared memory and multiplying them
  for( i = 0; i <= min_row_col; i += BLOCK_SIZE ) {
    L_block[ ty ][ tx ] = LU( row_adj, i + tx );
    U_block[ ty ][ tx ] = LU( i + ty, col_adj );

    __syncthreads();

    #pragma unroll( BLOCK_SIZE )
    for( k = 0; k < BLOCK_SIZE; ++k )
      sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];
    __syncthreads();
  }

  // Loops are unrolled by multiples of BLOCK_SIZE and the remaining elements are computed separately
  L_block[ ty ][ tx ] = LU( row_adj, min( i + tx, max_col ) );
  U_block[ ty ][ tx ] = LU( min( i + ty, max_row ), col_adj );

  __syncthreads();

  // Terminate threads that overreach the bounds as they have served their purpose of reading data
  if( row >= sec_end_row || col >= sec_end_col )
```

```
49      return;
50
51    for( k = 0; k < ty; ++k )
52      sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];
53
54    // Formula for U
55    // Do not check for division by zero as this kernel assumes that no bad ↩
          element is present in the diagonal section
56    sum = ( A( row, col ) - sum ) / LU( row, row );
57
58    // Check if the element (row, col) has been processed
59    // Read LU( row, col ) from shared memory instead of global memory
60    if( abs( U_block[ ty ][ tx ] - sum ) > process_tol )
61      processed( sec_id ) = false;
62
63    // Assign the element for the next iteration
64    LUnext( row, col ) = sum;
65  }
```

Listing C.4: The implementation of the `RSecCompute_kernel()` kernel which computes one iteration of a right section.

```
1  template< typename MatrixView, typename Index >
2  __global__
3  void
4  NonDiagSecAssign_kernel( MatrixView LU, MatrixView LUnext, const Index ↩
       sec_start_col, const Index sec_end_col, const Index sec_start_row, const ↩
       Index sec_end_row )
5  {
6    Index row = blockIdx.y * blockDim.y + threadIdx.y + sec_start_row;
7    Index col = blockIdx.x * blockDim.x + threadIdx.x + sec_start_col;
8
9    if( row >= sec_end_row || col >= sec_end_col )
10     return;
11
12   LU( row, col ) = LUnext( row, col );
13 }
```

Listing C.5: The implementation of the `NonDiagSecAssign_kernel()` kernel that assigns values of the next iteration to the matrix representing the current iteration.

Appendix D

SSPP Implementation

The implementation of SSPP is shown in Listing D.1.

```
template< typename Matrix, typename Vector >
void
SequentialSolver::solve( const Matrix& LU, Matrix& X, const Vector& piv )
{
  using Real = typename Matrix::RealType;
  using Device = TNL::Devices::Host;
  using Index = typename Matrix::IndexType;

  const Index num_rows = LU.getRows();
  const Index num_cols = LU.getColumns();

  const Index nrhs = X.getColumns();
  TNL::Containers::Vector< Real, Device, Index > sum( nrhs );

  // Solve (LU)X = B, where X holds the values of B on input

  // Order Matrix X according to vec
  if( ! piv.empty() ) {
    Decomposers::BaseDecomposer::orderMatrixAccordingTo( X, piv );
  }

  // Just to keep the original labels in the loops for clarity
  auto X_view = X.getView();
  auto Y_view = X.getView();
  auto B_view = X.getView();

  auto LU_view = LU.getConstView();
  auto sum_view = sum.getView();

  // Forward substitution: LY = B
  for( Index i = 0; i < num_rows; ++i ) {
    sum_view.setValue( 0 );
    for( Index j = 0; j < i; ++j ) {
      // sum = sum + LU( i, j ) * Y( j );
      auto y_row = Y_view.getRow( j );
      auto rhsSum = [ = ] __cuda_callable__( Index rhs ) mutable
      {
        sum_view( rhs ) = sum_view( rhs ) + LU_view( i, j ) * y_row.getValue(↩
            rhs );
      };
      TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, rhsSum );
    }
```

```
42      // Y( i ) = ( B( i ) - sum ) / LU( i, i );
43      auto y_row = Y_view.getRow( i );
44      auto b_row = B_view.getRow( i );
45      auto ySet = [ = ] __cuda_callable__( Index rhs ) mutable
46      {
47        y_row.setValue( rhs, ( b_row.getValue( rhs ) - sum_view( rhs ) ) / ↩
            LU_view( i, i ) );
48      };
49      TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, ySet );
50    }
51
52    // Backward substitution: UX = Y
53    for( Index i = num_rows - 1; i >= 0; --i ) {
54      sum_view.setValue( 0 );
55      for( Index j = i + 1; j < num_cols; ++j ) {
56        // sum = sum + LU( i, j ) * X( j );
57        auto x_row = X_view.getRow( j );
58        auto rhsSum = [ = ] __cuda_callable__( Index rhs ) mutable
59        {
60          sum_view( rhs ) = sum_view( rhs ) + LU_view( i, j ) * x_row.getValue(↩
              rhs );
61        };
62        TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, rhsSum );
63      }
64      // X( i ) = Y( i ) - sum;
65      auto x_row = X_view.getRow( i );
66      auto y_row = Y_view.getRow( i );
67      auto xSet = [ = ] __cuda_callable__( Index rhs ) mutable
68      {
69        x_row.setValue( rhs, y_row.getValue( rhs ) - sum_view( rhs ) );
70      };
71      TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, xSet );
72    }
73  }
```

Listing D.1: Excerpt from the implementation of SSPP. The code has been slightly modified for brevity, for example, the checks for appropriate sizing of matrices and vectors have been removed.

Appendix E

ISxPP Implementation

The implementation of IS_xPP is shown in Listing E.1. Note that the kernels called in Listing E.1 are shown separately in Listings E.2, E.3, and E.4.

```
template< const int THREADS_PER_BLOCK >
template< typename Matrix, typename Vector >
void
IterativeSolver< THREADS_PER_BLOCK >::solve( const Matrix& LU, Matrix& X, const Vector& piv )
{
  using Real = typename Matrix::RealType;
  using Index = typename Matrix::IndexType;

  const Index num_rows = LU.getRows();
  const Index num_cols = LU.getColumns();

  const Index nrhs = X.getColumns();

  // Solve (LU)X = B, where X holds the values of B on input

  // Order Matrix X according to vec
  if( ! piv.empty() ) {
    Decomposers::BaseDecomposer::orderMatrixAccordingTo( X, piv );
  }

  Matrix Xnext;
  Xnext.setLike( X );

  Matrix Y;
  Y.setLike( X );
  Matrix Ynext;
  Ynext.setLike( Y );

  // X holds the values of B on input
  auto X_view = X.getView();
  auto Y_view = Y.getView();
  auto B_view = X.getView();
  auto Xnext_view = Xnext.getView();
  auto Ynext_view = Ynext.getView();

  auto LU_view = LU.getConstView();

  // Flag to indicate that a section has been processed
  TNL::Containers::Array< bool, TNL::Devices::Cuda > processed{ 1, true };
  auto processed_view = processed.getView();
```

```
41   const Real process_tol = 0.0;
42
43   // Round to nearest higher multiple of THREADS_PER_BLOCK
44   Index num_rows_rounded = ( num_rows + THREADS_PER_BLOCK - 1 ) / ↩
         THREADS_PER_BLOCK * THREADS_PER_BLOCK;
45
46   // Number of right-hand sides is usually small
47   const Index BLOCK_SIZE_y = 8;
48   const Index nrhs_rounded = ( nrhs + BLOCK_SIZE_y - 1 ) / BLOCK_SIZE_y * ↩
         BLOCK_SIZE_y;
49
50   const Index blocks_perGrid_x = num_rows_rounded / THREADS_PER_BLOCK;
51   const Index blocks_perGrid_y = nrhs_rounded / BLOCK_SIZE_y;
52
53   dim3 threads_perBlock( THREADS_PER_BLOCK, BLOCK_SIZE_y );
54   dim3 blocks_perGrid( blocks_perGrid_x, blocks_perGrid_y );
55
56   // Forward substitution: LY = B
57   do {
58     processed_view.setElement( 0, true );
59     FowardSubst_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, ↩
           Y_view, Ynext_view, B_view, num_rows, nrhs, processed_view, ↩
           process_tol );
60
61     MtxAssign_kernel<<< blocks_perGrid, threads_perBlock >>>( Y_view, ↩
           Ynext_view, num_rows, nrhs );
62   } while( ! processed_view.getElement( 0 ) );
63
64   // Backward substitution: UX = Y
65   do {
66     processed_view.setElement( 0, true );
67     BackwardSubst_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, ↩
           X_view, Xnext_view, Y_view, num_rows, num_cols, nrhs, processed_view,↩
            process_tol );
68
69     MtxAssign_kernel<<< blocks_perGrid, threads_perBlock >>>( X_view, ↩
           Xnext_view, num_rows, nrhs );
70   } while( ! processed_view.getElement( 0 ) );
71 }
```

Listing E.1: Excerpt from the implementation of IS_*x*PP. The code has been slightly modified for brevity, for example, the checks for appropriate sizing of matrices and vectors have been removed. Note that the CUDA thread blocks used in the implementation are larger in the 1st dimension. Threads adjacent in the 1st dimension are assigned to neighboring elements in the same column since the matrices are stored in column-major order on the GPU. In other words, to mitigate misaligned global memory access, the 1st dimension of threads is used to access elements in a single column and the 2nd dimension is used to differentiate between right-hand sides.

```
1 template< typename ConstMatrixView, typename MatrixView, typename Index, ↩
     typename BoolArrayView, typename Real >
2 __global__
3 void
4 FowardSubst_kernel( const ConstMatrixView LU, MatrixView Y, MatrixView Ynext,↩
       const MatrixView B, const Index num_rows, const Index nrhs, ↩
     BoolArrayView processed, const Real process_tol )
5 {
6   // Each thread computes one element (row, rhs)
7   const Index row = blockIdx.x * blockDim.x + threadIdx.x;
8   const Index rhs = blockIdx.y * blockDim.y + threadIdx.y;
9
```

```
  // Terminate threads that overreach matrix bounds
  if( row >= num_rows || rhs >= nrhs )
    return;

  Real sum = 0;

  for( Index j = 0; j < row; ++j )
    sum = sum + LU( row, j ) * Y( j, rhs );

  sum = ( B( row, rhs ) - sum ) / LU( row, row );

  // Check if the element (row, rhs) has been processed
  if( abs( Y( row, rhs ) - sum ) > process_tol )
    processed( 0 ) = false;

  // Assign the element for the next iteration
  Ynext( row, rhs ) = sum;
}
```

Listing E.2: The implementation of the `FowardSubst_kernel()` kernel which computes one forward-substitution iteration.

```
template< typename ConstMatrixView, typename MatrixView, typename BoolArrayView, typename Real, typename Index >
__global__
void
BackwardSubst_kernel( const ConstMatrixView LU, MatrixView X, MatrixView Xnext, const MatrixView Y, const Index num_rows, const Index num_cols, const Index nrhs, BoolArrayView processed, const Real process_tol )
{
  // Each thread computes one element (row, rhs)
  const Index row = blockIdx.x * blockDim.x + threadIdx.x;
  const Index rhs = blockIdx.y * blockDim.y + threadIdx.y;

  // Terminate threads that overreach matrix bounds
  if( row >= num_rows || rhs >= nrhs )
    return;

  Real sum = 0;

  for( Index j = row + 1; j < num_cols; ++j )
    sum = sum + LU( row, j ) * X( j, rhs );

  sum = Y( row, rhs ) - sum;

  // Check if the element (row, rhs) has been processed
  if( abs( X( row, rhs ) - sum ) > process_tol )
    processed( 0 ) = false;

  // Assign the element for the next iteration
  Xnext( row, rhs ) = sum;
}
```

Listing E.3: The implementation of the `BackwardSubst_kernel()` kernel which computes one backward-substitution iteration.

```
template< typename MatrixView, typename Index >
__global__
void
MtxAssign_kernel( MatrixView M, MatrixView Mnext, const Index num_rows, const Index num_cols )
```

```
{
  // Each thread assigns one element (row, rhs)
  const Index row = blockIdx.x * blockDim.x + threadIdx.x;
  const Index rhs = blockIdx.y * blockDim.y + threadIdx.y;

  // Terminate threads that overreach matrix bounds
  if( row >= num_rows || rhs >= num_cols )
    return;

  M( row, rhs ) = Mnext( row, rhs );
}
```

Listing E.4: The implementation of the `MtxAssign_kernel()` kernel that assigns values of the next iteration to the matrix representing the current iteration.