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 Matthew 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 24.07.2023

.....

Bc. Lukáš Matthew Čejka

Poděkování

Chtěl bych poděkovat doc. Ing. Tomáši Oberhuberovi, Ph.D. za vedení mé diplomové práce a za podnětné návrhy, které ji obohatily. Dále bych rád poděkoval Ing. Jakubu Šístkovi, Ph.D. a Ing. Danielu Večerkovi za sdílené znalosti a výpomoc. 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. Additionally, I would like to express my gratitude to Ing. Jakub Šístek, Ph.D. and Ing. Daniel Večerka for sharing their knowledge and providing assistance. 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 Matthew Čejka

Název práce:

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

Autor: Bc. Lukáš Matthew 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: Tato diplomová práce se zaměřuje na implementaci iterativního paralelního LU rozkladu s tzv. pivotingem (LUP) na grafické karty (GPU) Nvidia. Primárním cílem této práce je aplikovat implementaci jako předpodmíňovač v existující knihovně a analyzovat její výkon. Za tímto účelem byl vyvinut projekt zahrnující postupy řešení soustav rovnic pomocí LUP spolu s jednotkovými a vykonnostními testy. Dále byl projekt integrován do víceúrovňové knihovny BDDC řešičů. Výkonnostní testy porovnávaly implementaci algoritmu s existujícími knihovnami GPU z hlediska výkonu a přesnosti na moderním výpočetním systému. Na základě komplexní analýzy výsledků výkonnostních testů byla vydána doporučení týkající se vhodnosti implementací pro obecné a účelové použití.

Klíčová slova: Paralelní LU rozklad s tzv. pivotingem, Iterativní metody, Grafická karta (GPU), Compute Unified Device Architecture (CUDA), Template Numerical Library (TNL), High-Performance Computing (HPC), Porovnání výkonu, Porovnání přesnosti, cuBLAS, cuSOLVER, BDDCML

Title:

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

Author: Bc. Lukáš Matthew Čejka

Abstract: This diploma thesis focuses on the implementation of an iterative parallel LU decomposition algorithm with Pivoting (LUP) for Nvidia Graphics Processing Units (GPUs). The primary objectives of this thesis are to apply the implementation as a preconditioner in an existing library and analyze its performance. For this purpose, a project was developed, encompassing procedures for solving systems of equations using LUP, along with unit tests and a benchmarking framework. Furthermore, the project was integrated into a multilevel BDDC solver library. The benchmarks conducted compared the implementation of the algorithm to established GPU libraries in terms of performance and accuracy on a state-of-the-art High-Performance Computing (HPC) cluster. Based on a comprehensive analysis of the benchmark results, recommendations were made regarding the suitability of the implementations for general use and specific problems.

Key words: Parallel LU decomposition with Pivoting, Iterative methods, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA), Template Numerical Library (TNL), High-Performance Computing (HPC), Performance comparison, Accuracy comparison, CuBLAS, cuSOLVER, BDDCML

Contents

Introduction

Solving scientific and engineering problems often involves employing various procedures to find solutions. One common approach is to formulate the problem as a system of equations, allowing the use of established methods. In practice, systems of linear equations arise in fields such as engineering, physics, economics, and computer science. On the other hand, systems of partial differential equations are prevalent in fields such as electrodynamics, fluid dynamics, thermodynamics, and diffusion. These systems can often be solved by representing them in matrix form and applying algebraic operations to find the solution.

There are numerous methods available for solving systems of equations, with Gaussian Elimination Method (GEM) and Lower-Upper decomposition with Pivoting (LUP) being among the most well-known. Using LUP to solve systems of equations involves two distinct stages: decomposition and substitution. In both stages, two main groups of numerical methods can be employed: direct or iterative. Direct methods arrive at the result in a finite number of steps, while iterative methods improve an initial estimate until a desired accuracy of the result is reached. However, direct methods are often sequential and difficult to parallelize, which limits their scalability on larger problems. In contrast, iterative methods can be effectively parallelized. Taking advantage of parallelized procedures becomes particularly beneficial for larger problems when utilizing Graphics Processing Units (GPUs), as they are designed to handle substantial concurrent workloads.

The primary objective of this thesis is to introduce and implement an iterative parallel LUP algorithm on the GPU and evaluate its performance. Additionally, this thesis presents a project that aims to facilitate the procedures used in both stages of solving systems of equations with LUP.

The first chapter introduces fundamental theoretical knowledge related to Nvidia GPUs and the application of LUP for solving systems of equations. In the second chapter, the developed project and other functionalities implemented to facilitate the use of the iterative algorithm are presented. Finally, the last chapter provides a comprehensive analysis in the form of a benchmark, evaluating the performance of the procedures available in the project.

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 on GPUs (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 certain tasks is not as straightforward, for example, Crout Method with partial pivoting (detailed in Section 1.3.2).

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

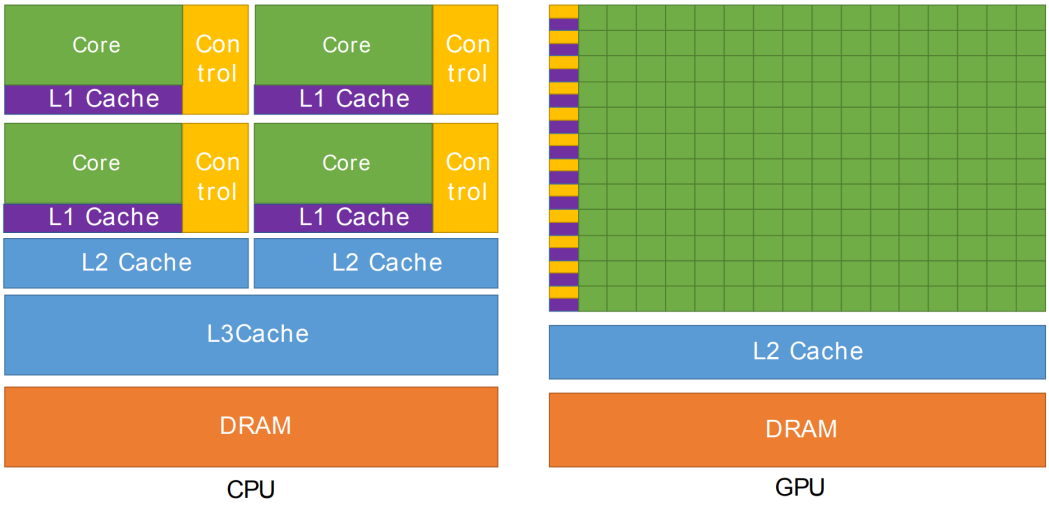


Figure 1.1: Architecture comparison of a CPU and a GPU. Taken from *CUDA C++ Programming Guide* [3].

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

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

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

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

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

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

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

1.2 Compute Unified Device Architecture (CUDA)

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

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

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

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

1.2.1 Fundamental Terminology

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

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

1.2.2 Thread Management

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

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

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

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

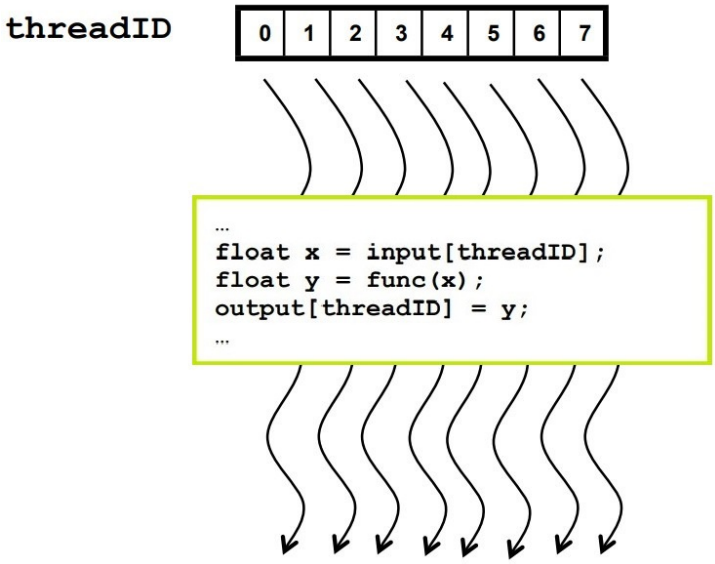


Figure 1.2: Execution of code by an 8-thread warp. The unique ID of each thread is stored in the variable `threadID`. In this example, `threadID` is used to read and write different data. Taken from *Getting Started with CUDA* [13] and *Parallel LU Decomposition for the GPU* [2].

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

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

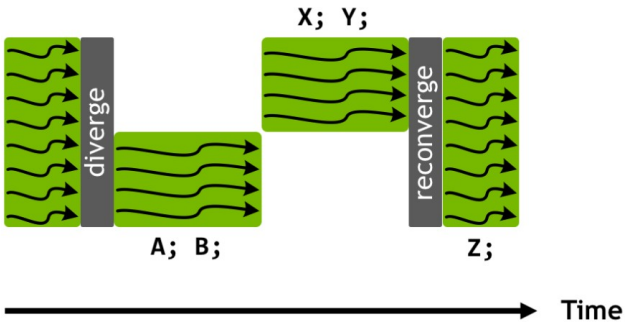


Figure 1.3: Pseudocode showcasing thread divergence in an 8-thread warp. The `threadIdx.x` variable contains each thread's ID in the 1st dimension. In this example, while threads `0` to `3` execute statements `A` and `B`, threads `4` to `7` are idle. Then, threads `4` to `7` execute statements `X` and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

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

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

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

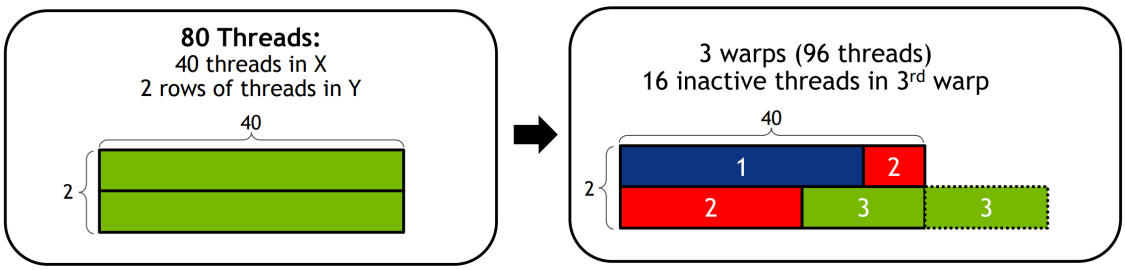


Figure 1.4: Division of threads into warps in a block of 40 by 2 threads. Each green rectangle in the left image represents a row of threads. In the right image, each color represents a unique warp. Taken from *VOLTA Architecture and performance optimization* [18].

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

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

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

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

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

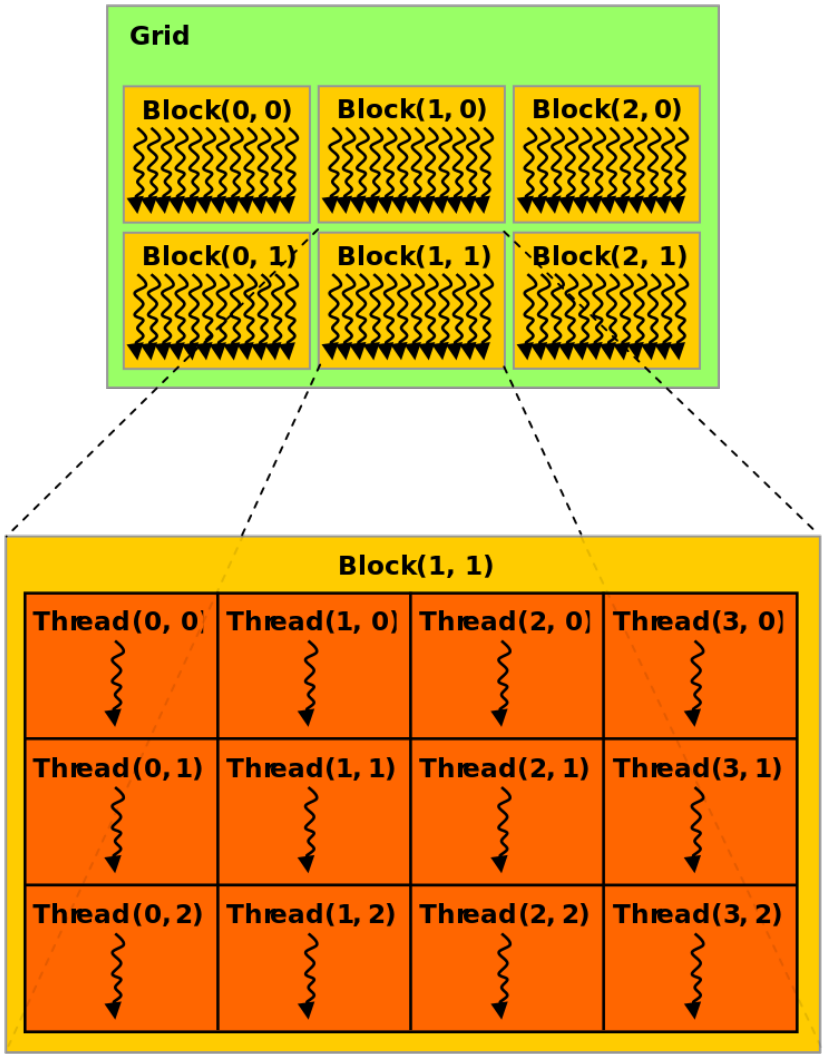


Figure 1.5: Visualization of a 2-by-3 grid comprising 3-by-4 thread blocks. The numbers in the brackets are the IDs of each entity in each dimension. Taken from *CUDA C++ Programming Guide* [3].

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

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

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

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

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

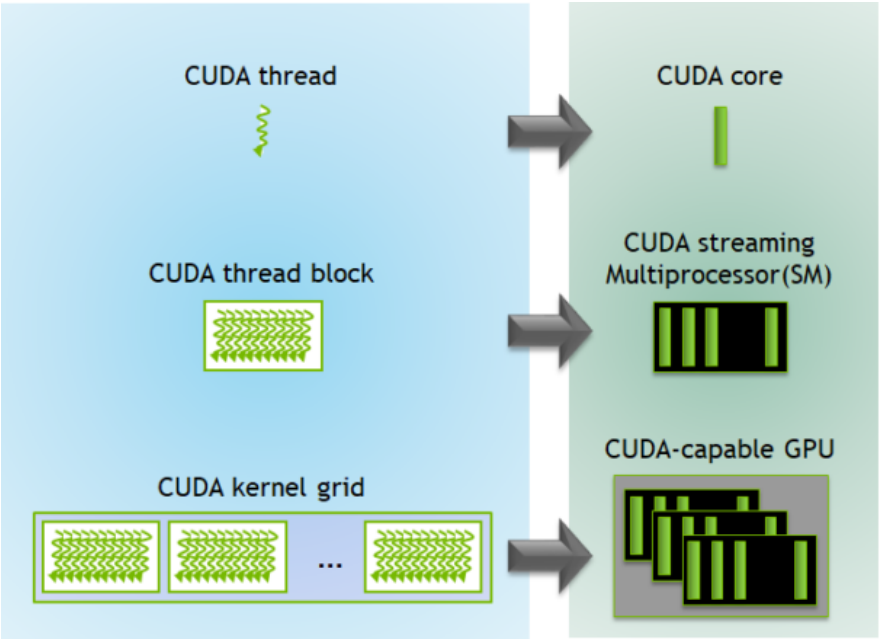


Figure 1.6: Visualization illustrating the execution of CUDA thread structures by different hardware components in an Nvidia GPU. Taken from *CUDA Refresher: The CUDA Programming Model* [20].

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

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

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

Listing 1.2: Excerpt of C++ code portraying the creation of a grid configuration and the launching of a kernel. The example implementation 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(...)` kernel 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.

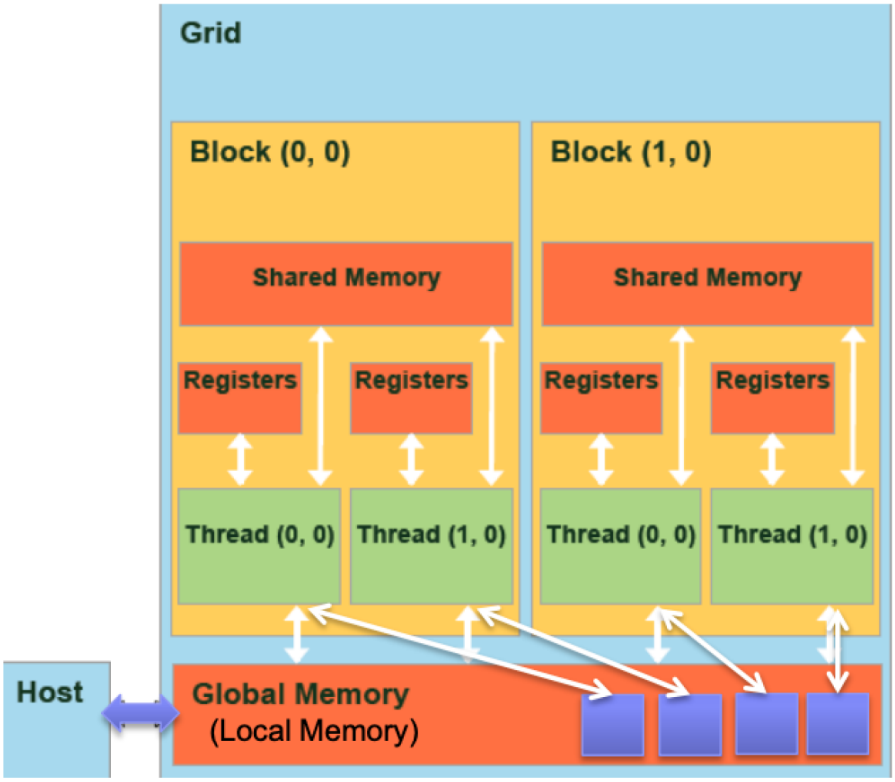


Figure 1.7: CUDA memory structuring with accesses available to each thread structure. Taken from *GPU* by Hsiao, Y. [22].

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

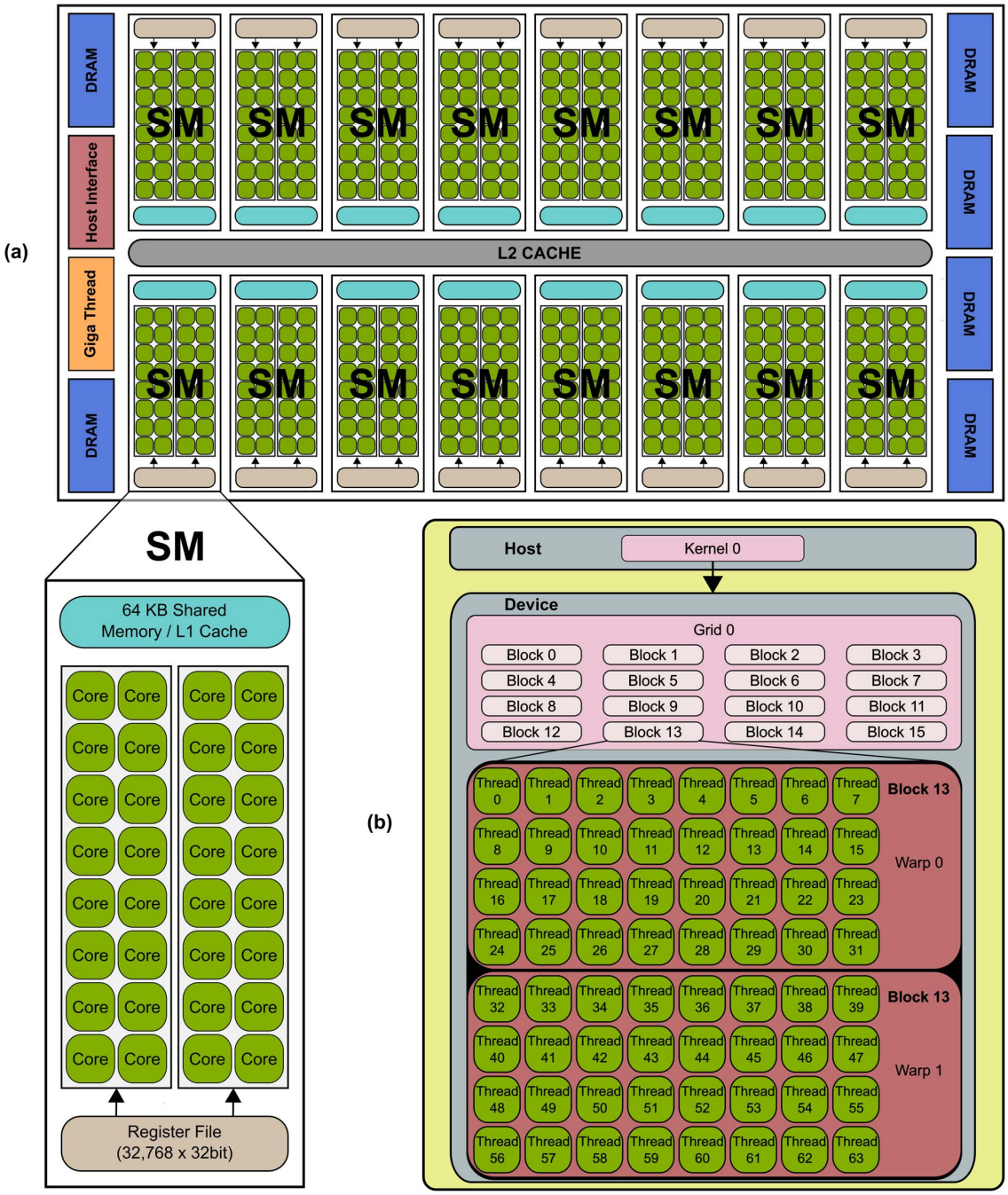


Figure 1.8: Simplified overview of an Nvidia GPU's architecture and the CUDA programming model. Taken from *Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs* [23].

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 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.

```
1 // Declare array of 2 stream objects
2 cudaStream_t streams[ 2 ];
3
```

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

```
4  // Create each stream
5  for( int i = 0; i < 2; ++i ) {
6    cudaStreamCreate( &streams[ i ] );
7  }
8
9  // Launch MyKernelA using the 0th stream on a grid of one one-thread block ↩
       with 0 bytes of dynamically allocated shared memory
10 MyKernelA<<< 1, 1, 0, stream[ 0 ] >>>( inputVariableA )
11
12 // Launch MyKernelB using the 1st stream
13 MyKernelB<<< 1, 1, 0, stream[ 1 ] >>>( inputVariableB )
14
15 // Destroy each stream
16 for( int i = 0; i < 2; ++i ) {
17   cudaStreamDestroy( stream[ i ] );
18 }
```

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.

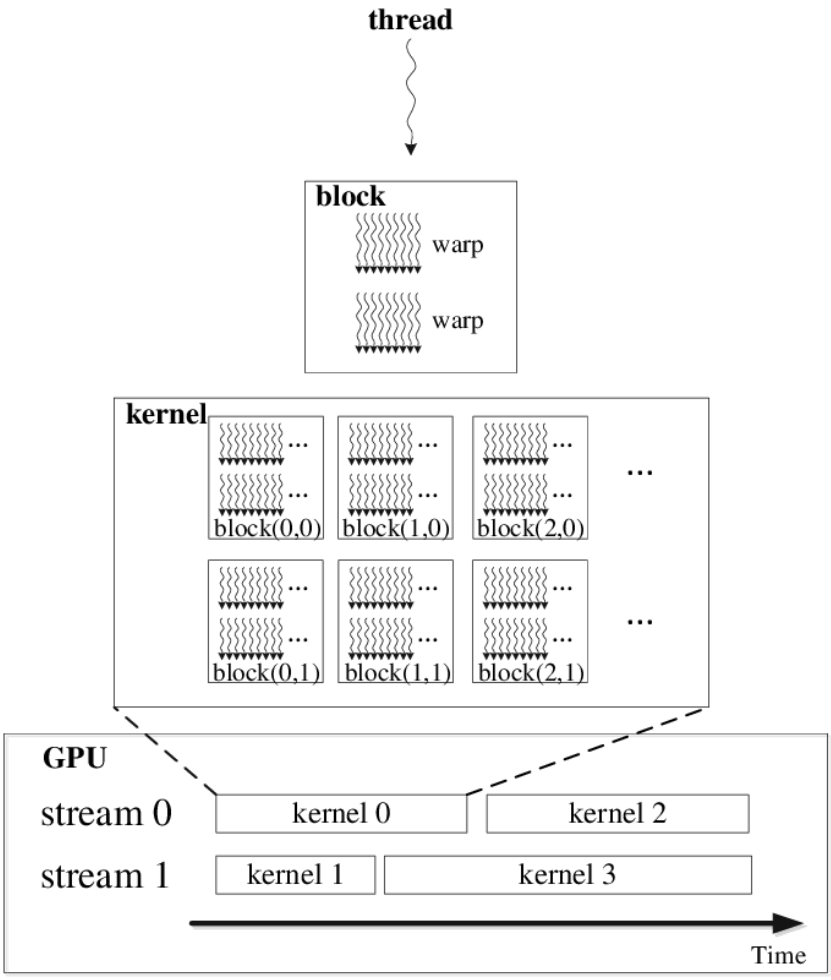


Figure 1.9: Visualization of concurrent kernel execution. Taken from *POM.gpu-v1.0* [26].

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 certain 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.

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 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].

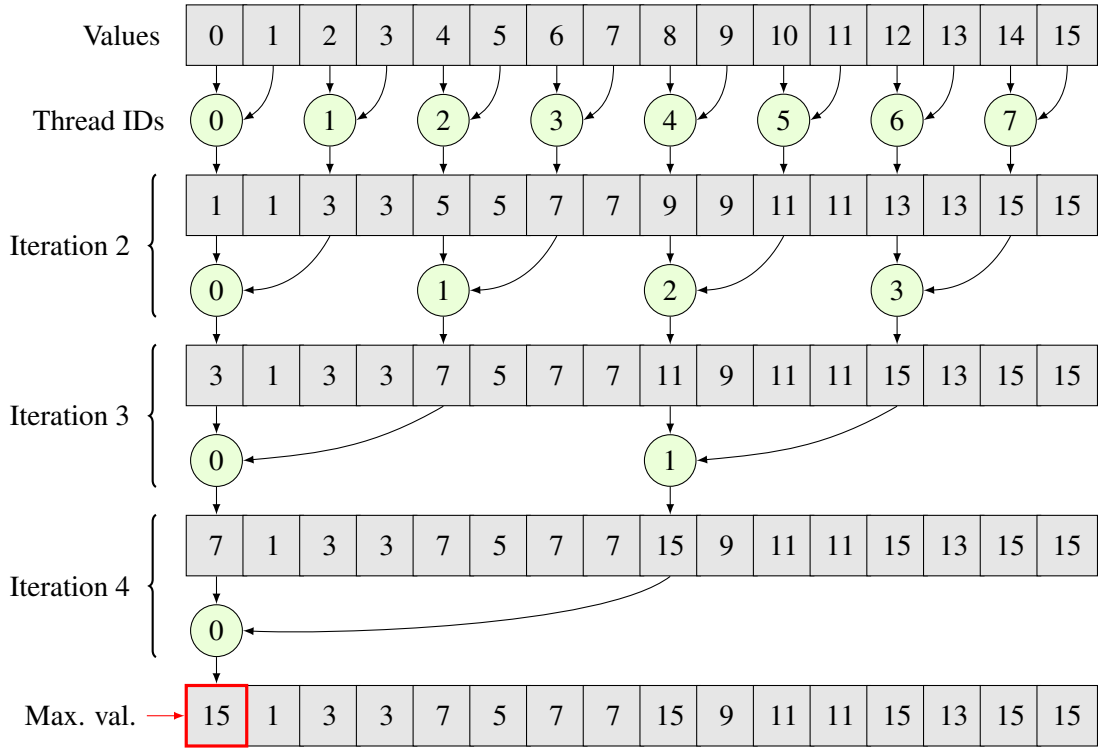


Figure 1.10: Visualization of finding the maximum value in a 16-element array stored in shared memory using parallel reduction. The gray squares contain values of the array. Each row of gray squares shows the array contents in a given iteration. The light green circles represent threads; the number inside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

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 include, 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 efficiently perform fundamental tasks, such as solving a system of linear equations.

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 Durand Crout's matrix decomposition method, referred to - in the context of this project - as the *Iterative Crout's 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.

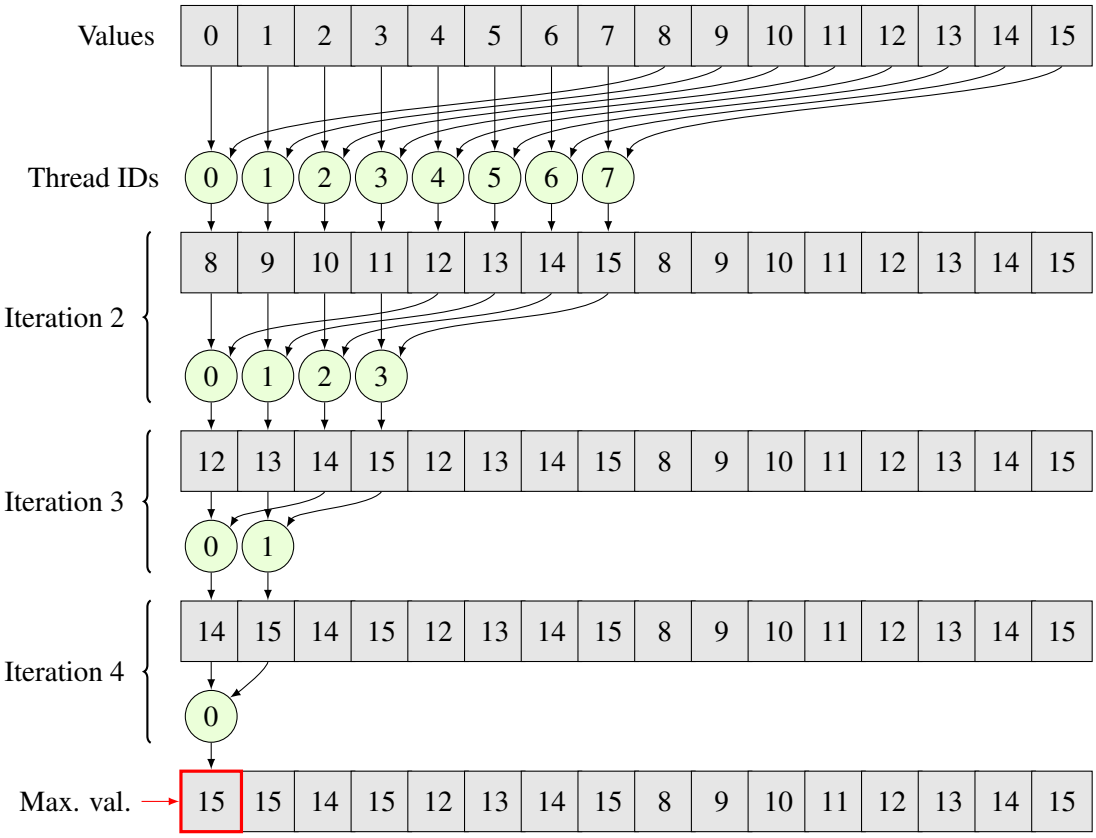


Figure 1.11: Visualization of finding the maximum value in a 16-element array stored in shared memory using parallel reduction with *sequential addressing*. The gray squares contain values of the array. Each row of gray squares shows the array contents in a given iteration. The light green circles represent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

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 $\mathbf{A}$, which is referred to as *LU decomposition with full pivoting*. Furthermore, the factorization is numerically stable in practice even if only rows are permuted [31], which is 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 using **P** 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 linear 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, 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.

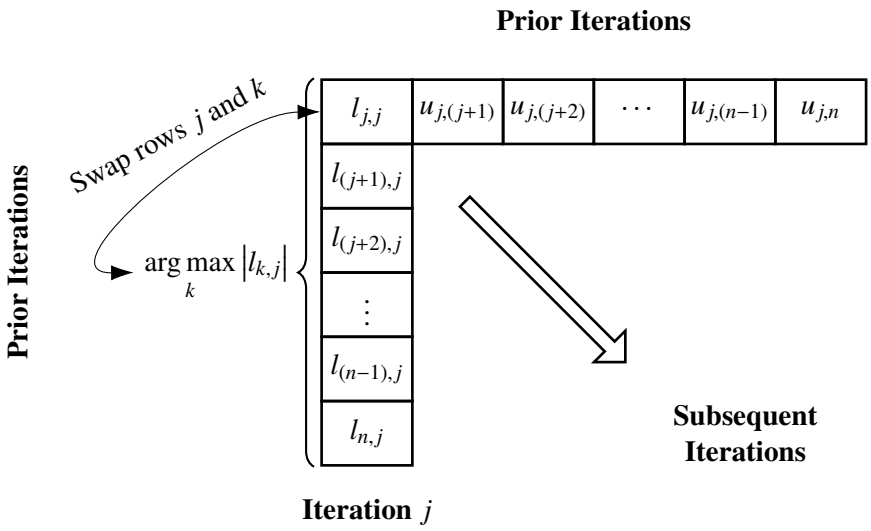


Figure 1.12: Visualization of the advance of CMPP's algorithm. First, elements $l_{j,j}$ to $l_{n,j}$ in column j of $\mathbf{L}$ are computed. Then, row j is pivoted. Its row is swapped with the row containing the largest absolute value of an element in column j, starting from $l_{j,j}$ and ending at $l_{n,j}$, both inclusive. Finally, elements $u_{j,j+1}$ to $u_{j,n}$ in row j of $\mathbf{U}$ are computed. Adapted from *Crout's LU Factorization* by Vismor [36].

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

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

Listing 1.5: C++ pseudocode for CMPP's algorithm that decomposes 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 **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 zeros. 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 certain 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.

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.

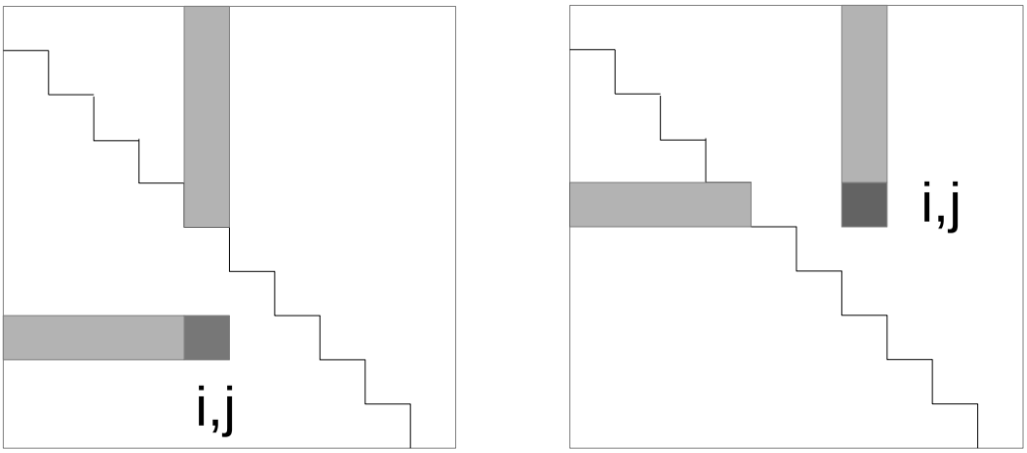


Figure 1.13: Two examples of elements used (light gray squares) to compute the sum in the formula of elements $l_{i,j}$ and $u_{i,j}$ (dark gray squares). The lower triangle of the matrix consists of the corresponding elements of the lower triangle from $\mathbf{L}$ while the upper triangle consists of the corresponding elements of the upper triangle from $\mathbf{U}$. 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].

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 $\mathbf{A} \in \mathbb{R}^{n \times n}$ ($n \in \mathbb{N}$) 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 $\mathbf{A}$:

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

   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:

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

4. Take the first $j \in \widehat{n}$ that satisfies the conditions $\left|l^{t-1}_{j,j} - l^t_{j,j}\right| \leq \epsilon_1$ (where ϵ_1 denotes a tolerance, e.g., 0.001) and $\left|l^t_{j,j}\right| \leq \epsilon_2$, where ϵ_2 denotes a *pivoting tolerance*, e.g., 1e-5. In other words, find the first element on the main diagonal of $\mathbf{L}$ whose value is finalized and check if it requires pivoting according to the pivoting tolerance. In the context of this project, pivoting an element only when it is smaller than (or equal to) the pivoting tolerance is referred to as *conditional partial pivoting*. If no such j satisfies the conditions, proceed to Step 6. Then, *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 $1 \leq c \leq j$. In this context, *iteratively converge* signifies to continue computing the mentioned elements using Formula 1.11 until they all satisfy the condition $\left|l^{t-1}_{b,c} - l^t_{b,c}\right| \leq \epsilon_1$.

5. Pivot row j (same as Step 4 in the CMPP algorithm described on Page 32):

   (a) Find the index (p) of the element largest in absolute value in column j from row j to n:

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

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

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

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_1, then increment t by 1 and return to Step 3.

8. The algorithm has converged to an approximate solution of $\mathbf{A} = \mathbf{LUP}$, where $\mathbf{A}$ represents its original state before any potential changes occurred during this algorithm.

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 conditions 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, in Section 1.3.3, 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 parallel 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, 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 website URL: `https://tnl-project.org`
[2] cuSOLVER website URL: `https://developer.nvidia.com/cusolver`
[3] cuBLAS website 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 various 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 those 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. However, it is noteworthy that TNL's parallelized for-loop cannot be run within kernels as it does not support dynamic parallelism.

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 `cusolverDnXgetrs()` 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 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 $\mathbf{LY} = \mathbf{B}$, where **Y** and **B** are appropriately-sized matrices. Note that **B** is expected to be permuted using **P** before **B** is supplied to `cublas<t>trsm()`.
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 $\mathbf{LY} = \mathbf{B}$ 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 `cublasDtrsm()` 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.
- 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.

[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`

- 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., $\mathbf{A} = \mathbf{LUP}$, or LU decomposition without partial pivoting, i.e., $\mathbf{A} = \mathbf{LU}$. 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 employs *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 $\mathbf{A}$, and `piv` is expected to be appropriately sized. 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 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 either `TNL::Matrices::DenseMatrix` and `TNL::Containers::Vector`, respectively, or types that inherit from them.

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

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 $\mathbf{L}$ and $\mathbf{U}$ have been successfully computed.
3. Compute values from $l_{j,j}$ to $l_{n,j}$ in column j of $\mathbf{L}$ 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 **L**, **U**, and **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 **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 **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 ICMPP 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:

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

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.

```
1 template< typename Matrix, typename Vector >
2 static void
3 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-by-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` if the elements they depend on have not been processed. The elements that an element of `LU` depends on are described as its *chains of dependencies*. The chains of dependencies for a specific element are visualized in Figure 2.1.

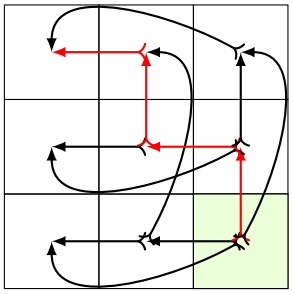


Figure 2.1: Visualization of the chains of dependencies for element `LU[ 2 ][ 2 ]` (green box) in a 3-by-3 matrix `LU`. Each arrow points to the element that the element it originates from requires for its computation. For example, to compute `LU[ 2 ][ 2 ]`, the elements it points to are required, i.e., `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. Note that the variant of conditional partial pivoting used in ICM_*x*PP differs from the variant used in CMPP and PCM_*x*PP. For the ICM_*x*PP decomposer, the tolerance of conditional partial pivoting is defined as:

$$\epsilon_2 < \left|l_{j,j}\right| \leq \epsilon_3 \,, \tag{2.6}$$

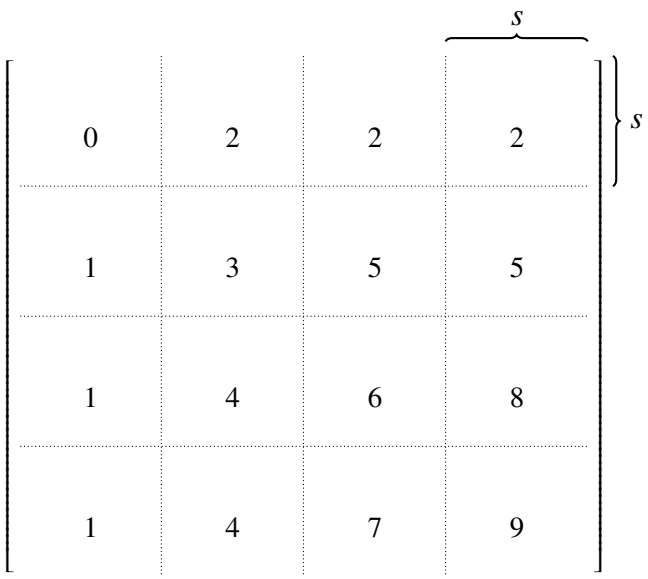


Figure 2.2: Visualization of matrix `LU` divided into sections of size s-by-s. The number inside each section indicates the order in which it is processed. First, the top-left section, labeled as section "0", is processed. Then, the sections below it, i.e., sections labeled with a "1", are processed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

where ϵ_2 denotes the lower bound of the tolerance interval and ϵ_3 denotes the inclusive upper bound. By default, ϵ_2 is set to 1e-5 and ϵ_3 is set to 1e+5. If the absolute value of $l_{j,j}$ falls outside this interval, it is considered a *bad element* that requires pivoting. The upper bound was introduced for ICM_*x*PP as a result of extensive testing performed on a set of matrices with varying characteristics. In some cases, without the upper bound, the ICM_*x*PP decomposer produced matrices with `NaN` values. This issue is discussed further in Section 3.1.3 on Page 87.

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 new bad element is present between `badEl_searchStart` and the bottom-right-most element of the diagonal section. If a new bad element is found, update 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, pivot the row containing the bad element. If the new value is zero or `NaN`, then the decomposition has failed. If the new value is a nonzero number but breaks the pivoting tolerance defined in Equation 2.6, update the value of `badEl_searchStart` to the next element on the diagonal so the computation can continue. This ensures that the bad element is avoided in the subsequent 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.

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

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

  Index sec_size = min( max( num_cols / 10, (Index) 256 ), (Index) 1024 );
  sec_size = ( sec_size + BLOCK_SIZE - 1 ) / BLOCK_SIZE * BLOCK_SIZE;

  // ...
}
```

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-by-300 for `BLOCK_SIZE` equal to 8 or 16, and 320-by-320 for `BLOCK_SIZE` equal to 32. However, if `LU` is a 12,000-by-12,000 matrix, then the size of each section is adjusted to 1024-by-1024 for all `BLOCK_SIZE` values.

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 - lower bound
const Real piv_tol_lower = 1.0e-5;
// Pivoting tolerance - uppder bound
const Real piv_tol_upper = 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_lower || piv_tol_upper < abs( LU_view( col, col ) ) ) ? 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 );

```

```
51      do { // Process the diagonal section
52        // Reset the processed flag
53        processed_view.setElement( 0, true );
54
55        // Compute the values up to and including the bad element
56        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 );
57
58        // Assign the values computed for the next iteration
59        DSecAssign_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, ↩
              LUnext_view, dSec_start, dSec_end, dSec_start, dSec_end, badEl_idx ↩
              );
60
61        // Check if a bad element is present in a different column
62        badEl_idx = TNL::Algorithms::reduce< TNL::Devices::Cuda >( ↩
              badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, ↩
              num_cols );
63      } while( ! processed_view.getElement( 0 ) );
64
65      // ... Processing of the sections below the diagonal section
66      // ... Pivoting of the bad element
67    } while( badEl_idx < dSec_end - 1 );
68    // ... Processing of the sections to the right of the diagonal section
69  }
```

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 49 and 62. 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 31 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 53 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 63, 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 zero. The value was chosen to provide accurate results as even small nonzero values, e.g., 1e-5, 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.

```
1  // ...
2  // Allocate and initialize an array of stream handles to process the sections in parallel
3  const Index nonDiagSecs_perRow = TNL::ceil( (double) num_cols / (double) sec_size ) - 1;
4  auto* streams = (cudaStream_t*) malloc( nonDiagSecs_perRow * sizeof( cudaStream_t ) );
5  for( Index i = 0; i < nonDiagSecs_perRow; ++i )
6    cudaStreamCreate( &( streams[ i ] ) );
7
8  // Flags indicating the processing status of non-diagonal sections for the device and the host
9  TNL::Containers::Array< bool, TNL::Devices::Cuda, Index > nonDiagSec_processed{ nonDiagSecs_perRow, false };
10 // The host array is used to avoid copying individual processing variables between the host and the device
11 TNL::Containers::Array< bool, TNL::Devices::Host, Index > nonDiagSec_processed_host{ nonDiagSecs_perRow, false };
12 // ...
13 auto nonDiagSec_processed_view = nonDiagSec_processed.getView();
14
15 // Fill the pivoting vector with increments of 1 starting from 1 to num_rows.
16 BaseDecomposer::setDefaultPivotingValues( piv );
17 auto piv_view = piv.getView();
18 // ...
19
20 // Non-diagonal section start, end, and ID (used to access the streams)
21 Index sec_start, sec_end, sec_id;
22
23 // Loop through the diagonal sections
24 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 ) )
25 {
26   do { // Process the diagonal section and the sections below it
27     // ... Processing of the diagonal section
28
29     // The Diagonal section contains processed values - excluding the values to the bottom-right of the bad element
30     // Compute the lower sections up to and including the column containing the bad element
```

```
31
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 own 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_lower, piv_tol_upper )↩
            ;
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
```

```
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;

  // Adjust 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
```

```
57   if( row >= sec_end_row || col >= sec_end_col )
58     return;
59
60   // 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 from global memory to shared memory, 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.

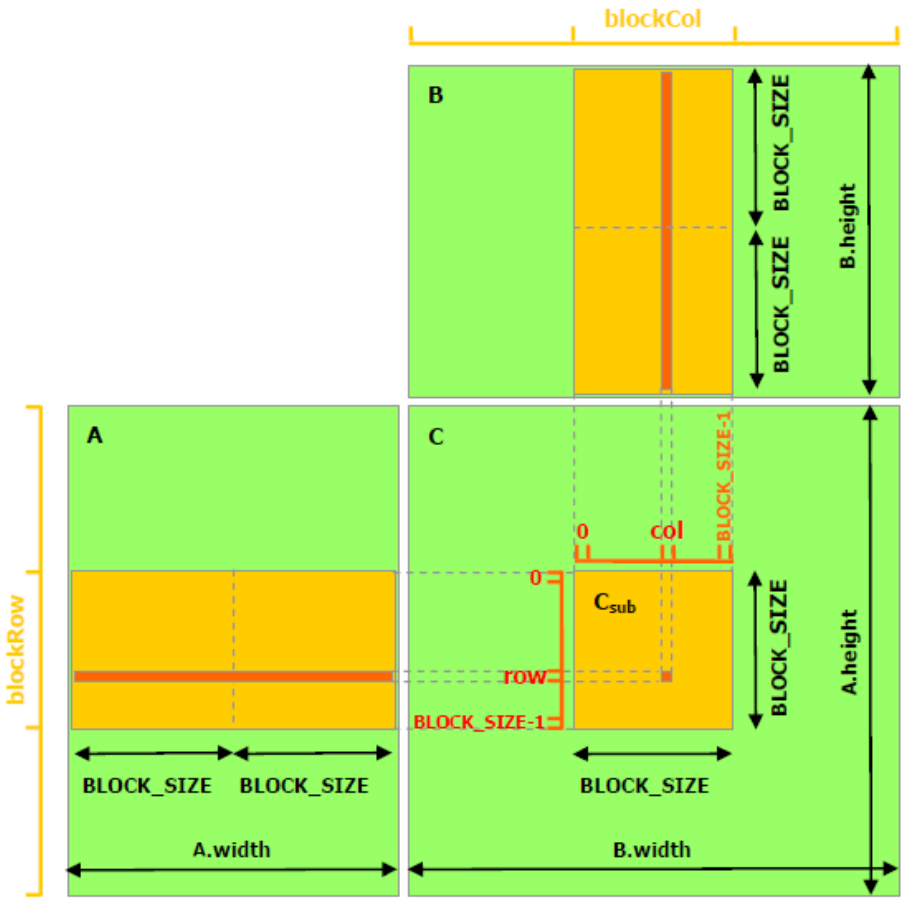


Figure 2.3: Visualization of matrix multiplication by blocks of elements using shared memory. Note that this figure, taken from *CUDA C++ Programming Guide* [3], depicts regular matrix multiplication, i.e., not the limited rendition present in Crout's method. The implementation in Listing 2.9 performs 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 the kernel computing 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-by-m matrix of unknowns, $\mathbf{P}$ is an n-by-n permutation matrix, and $\mathbf{B}$ is an n-by-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 is expected 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
CuSolverDnXgetrf(PP)	No	No	Yes	Yes
ICM_*x*(PP)	Yes	Yes	Yes	No
PCM_*x*(PP)	Yes	Yes	Yes	No

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 the decomposers and solvers implemented by the author of this thesis with those 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 absolute difference between the actual results and the 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 the `src/Benchmarks/utils` directory found in the Decomposition repository[16].

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

$$\max |\mathbf{A} - \mathbf{LUP}| \,, \tag{2.7}$$

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

For solvers the maximum absolute difference is defined as:

$$\max |\mathbf{LUPX} - \mathbf{B}| \,, \tag{2.8}$$

[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`

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

where matrices **L**, **U**, and **P** are the same as in Equation 2.7, **X** is the computed matrix of unknowns, and **B** is the matrix of right-hand sides. To clarify, the computed unknowns are used in the system of 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[17], 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[17] or in its GitHub repository[18].

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[19] 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.

[17] BDDCML website URL: `https://users.math.cas.cz/~sistek/software/bddcml.html`
[18] Multilevel BDDC Solver Library GitHub repository URL: `https://github.com/sistek/bddcml`
[19] 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.

Chapter 3

Comparing Decomposers and Solvers

This chapter presents and analyzes the results of benchmarks that were run to compare decomposers and solvers made available by the Decomposition project. The procedures were compared in two scenarios: raw performance and usability in an existing project. To achieve the former, the benchmark implemented in the Decomposition project, mentioned in Section 2.2.3, was used. To achieve the latter, the procedures were compared as part of BDDCML, mentioned in Section 2.3.

The source code used to run the benchmarks, i.e., the source code of the Decomposition project, TNL, tnl_bddcml_interface, and BDDCML, is available on request or as an attachment to this thesis. Additionally, the full logs are also available on request or as an attachment to this thesis.

First, the results of the Decomposition project benchmarks are presented. Then, the analysis of results obtained from the BDDCML benchmark, along with other performance-comparing efforts, is provided. Finally, a summary of the comparisons across all benchmarks is presented.

3.1 Decomposition Project Benchmarks

This section presents the results of the benchmarks implemented in the Decomposition project. The benchmarks, presented in Section 2.2.3, consist of running a procedure (`decompose()` of a decomposer or `solve()` of a solver) on an input matrix and recording various statistics, for example, the time of execution, the accuracy of the results, etc.

In terms of both the decomposers and solvers, only the variants with partial pivoting were compared. Benchmark results without partial pivoting are presented and analyzed in Chapter 3 of *Parallel LU Decomposition for the GPU* [2].

First, the specifications of the platform on which the benchmarks were run are presented. Next, the matrices used in the benchmarks are shown. Following that, the benchmark results of the decomposers are analyzed, and the benchmark results of the solvers are presented. Finally, a summary of the benchmark results is provided, along with recommendations.

3.1.1 Benchmark Platform Specifications

The benchmarks were run on the HPC (High-Performance Computing) platform of CTU referred to as the RCI (Research Center for Informations in CTU Prague) Cluster[1]. The cluster uses

[1]RCI website URL: `https://rci.cvut.cz/`

the SLURM[2] cluster management tool to schedule jobs on CPU and GPU compute nodes. The Decomposition benchmarks were run on a node with an AMD CPU and an Nvidia GPU. Specifically, the hardware and software configuration used for the benchmarks is presented in Table 3.1.

Type	Component	Specifications
Hardware	CPU	AMD EPYC 7543 @ 2.8GHz (32 cores, 64 threads)
	Memory	32GB RAM
	GPU	Nvidia Tesla A100 40GB HBM2
Software	OS	Red Hat 8.5.0-16
	CMake	3.24.3
	GCC	12.2.0
	CUDA	12.0.0

Table 3.1: Hardware and software configuration of the RCI cluster node that the benchmarks were run on. Taken from *RCI Cluster Wiki* [52] and the configuration of the computation job. The configuration is specified in a *batch* script which comprises SLURM-specific job settings along with Bash code for the compute node to execute. The scripts used for the benchmarks are available on request or as an attachment to this thesis.

3.1.2 Matrices Used for Benchmarks

The benchmarks for the procedures were run on a set of 50 matrices with varying characteristics. The reason behind the limited number of matrices is the running time of the benchmark comparing decomposers. Specifically, given the 50 matrices, the job responsible for benchmarking decomposers took between 68 and 72 hours to complete with the latter being the limit for jobs scheduled by RCI collaborators.

The set of matrices is made up of 28 sparse matrices and 22 dense matrices. The sparse matrices were obtained from *The university of Florida sparse matrix collection* [53] and the dense matrices either came from the "Poisson equation on a cube" problem in BDDCML, or they were randomly generated by a Python script presented in Appendix F. Note that the sparse matrices were stored as dense matrices using `TNL::Matrices::DenseMatrix` due to the lack of support for sparse matrix structures in the Decomposition project. The matrix sizes in the set ranged from 4-by-4 to 11,532-by-11,532.

From the set of 50 matrices, 15 were selected for the direct comparison of decomposers. The selected matrices were chosen to cover a diverse range of characteristics, for example, the size, density and sparsity, nonzero element structure, etc. The list of the 15 selected matrices can be found in Table 3.2.

3.1.3 Decomposers Benchmark

This section presents benchmark results obtained by decomposing the set of matrices using the selection of decomposers listed in Table 3.3.

As previously mentioned in Section 2.2.3, each matrix was decomposed 10 times by each decomposer. The metrics collected across the 10 runs were then averaged and logged. The benchmark was run for both double and single precision.

[2]Slurm website URL: `https://slurm.schedmd.com/overview.html`

Matrix	Rows/Columns	Nonzeros	Avg. nonzeros per row
poc-8_4_2-1	230	28,960	125.9
Cejka558	558	311,090	557.5
poc-24_2_2	1,814	2,749,120	1515.5
Cejka2599	2,599	6,753,502	2598.5
poc-32_2_2	3,182	8,868,688	2787.1
c-22	3,792	28,870	7.6
Cejka4156	4,156	17,270,197	4155.5
freeFlyingRobot_9	4,778	39,964	8.4
Cejka6574	6,574	43,214,237	6573.5
fp	7,548	834,222	110.5
Cejka8385	8,385	70,304,004	8384.5
nd3k	9,000	3,279,690	364.4
Cejka10135	10,135	102,713,214	10134.5
msc10848	10,848	1,229,776	113.4
sinc15	11,532	551,184	47.8

Table 3.2: Subset of 15 selected matrices used for the direct comparison of decomposers. The dense matrices randomly generated by the aforementioned Python script follow the naming pattern *Cejka<num_rows>* and the dense matrices arising from BDDCML are prefixed with *poc*. The remaining matrices are sparse and were obtained from *The university of Florida sparse matrix collection* [53].

Decomposer		Variants	Performer
Name	Abbreviation		
Crout's Method with Partial Piv.	CMPP	-	CPU
CuSolverDnXgetrf with Partial Piv.	CuSolverDnXgetrfPP	-	GPU
Iterative Crout's Method with Partial Piv.	ICM_*x*PP	8, 16, 32	GPU
Parallel Crout's Method with Partial Piv.	PCM_*x*PP	8, 16, 32	GPU

Table 3.3: Decomposers compared in the benchmark: CMPP (described in *Crout's Method with Partial Pivoting (CMPP)* in Section 2.2.2), CuSolverDnXgetrfPP (mentioned in Section 2.1.2), ICM_*x*PP (described in *Iterative Crout's Method with Partial Pivoting (ICM_xPP)* in Section 2.2.2), and PCM_*x*PP (described in *Parallel Crout's Method with Partial Pivoting (PCM_xPP)* in Section 2.2.2). The *Variants* column indicates the different configurations of the decomposer. For example, ICM_*x*PP was benchmarked as three separate decomposers depending on the CUDA thread block configuration, e.g., 8-by-8 threads per block, 16-by-16 threads per block, etc. Furthermore, for ICM_*x*PP, the processing tolerance was set to zero, and the lower bound and upper bound of conditional partial pivoting tolerance were set to 1e−5 and 1e+5, respectively. For CMPP and PCM_*x*PP, the conditional partial pivoting tolerance was set to 1e−5. In the case of PCM_*x*PP, the thread blocks are one-dimensional and the number of threads per block is x^2, i.e., 64, 256, and 1024. The *Performer* column specifies the primary device used by each decomposer.

One of the tasks for this thesis was to compare the implemented decomposers to a CPU library and a GPU library. However, the former was intentionally omitted as the performance of the implemented decomposers was severely lacking. Furthermore, neither PCM_*x*PP nor ICM_*x*PP were implemented for the CPU, and the implementation of the CMPP decomposer was not comparison-worthy. Thus, combined with the fact that the AMD-CPU-specialized library, *AOCL-ScaLAPACK*[3], which

[3]AOCL-ScaLAPACK website URL: `https://www.amd.com/en/developer/aocl/scalapack.html`

provides the LU factorization function, was not available in the RCI cluster, it was decided to forego implementing another wrapper into the Decomposition project to facilitate the comparison.

First, for both double and single precision, the execution times of the decomposers on the subset of 15 matrices are compared. Then, the comparison of the execution times on the entire set of 50 matrices is presented. Finally, the accuracy of the results is discussed.

Comparison of Execution Times on the Subset of Matrices

To compare the performance of the decomposers listed in Table 3.3 on the matrices mentioned in Table 3.2, the execution times of the decomposers are presented in this section. The comparison using both double and single precision can be found in Figure 3.1.

CuSolverDnXgetrfPP As can be seen from Figure 3.1, CuSolverDnXgetrfPP was the best-performing decomposer on the subset of matrices with both double and single precision. Furthermore, Figure 3.1 shows that, compared to the other decomposers, the execution times of CuSolverDnXgetrfPP increased at a smaller rate.

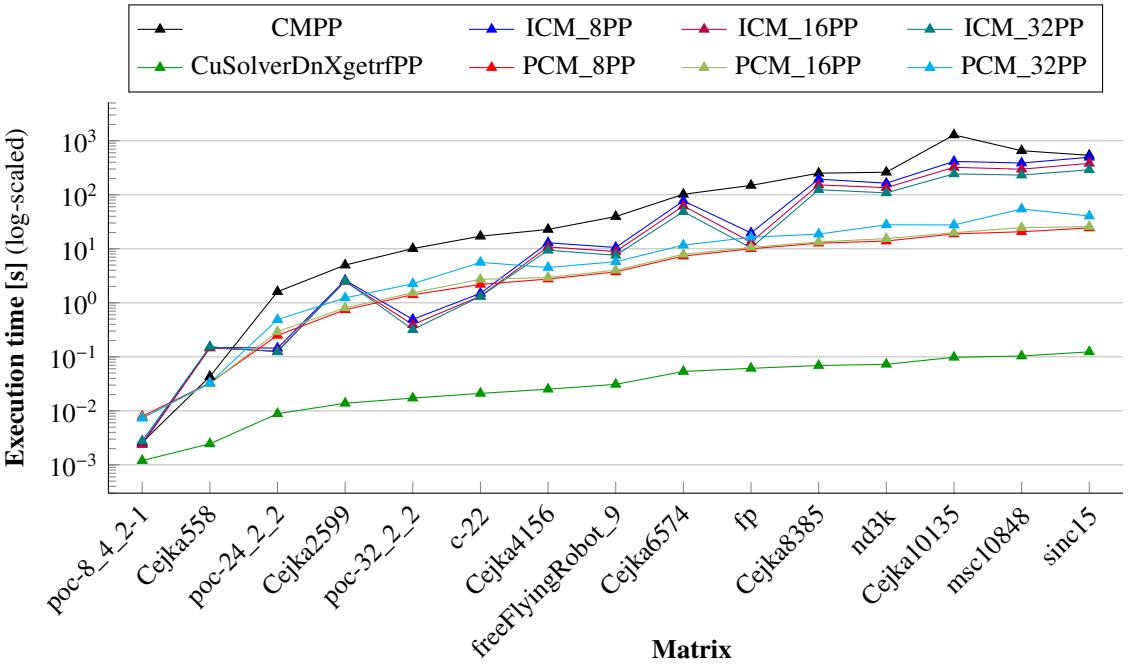
For context, when double precision was used, CuSolverDnXgetrfPP was able to decompose the *sinc15* matrix in 0.123 seconds, whereas the second fastest decomposer for this matrix, PCM_8PP took 24.349 seconds. However, CuSolverDnXgetrfPP was not able to maintain this lead when using single precision as its execution time decreased to only 0.108 seconds, while PCM_8PP decreased its time to 12.667. On the other hand, the third fastest decomposer, ICM_32PP, took 290 seconds to decompose the *sinc15* matrix using double precision and failed to decompose it using single precision.

ICM_*x*PP From Figure 3.1, it can be seen that the performance of ICM_*x*PP is not as consistent as that of PCM_*x*PP. Specifically, ICM_*x*PP consistently performed poorly when decomposing the *Cejka* dense matrices as all its variants were slower than every decomposer except for CMPP.
For a better understanding of why ICM_*x*PP does not perform well on randomly generated dense matrices, the nonzero element pattern of the *Cejka558* matrix along with the iterative and pivoting metrics of ICM_32PP are shown in Figure 3.2. Note that the iterative and pivoting metrics were near-identical across all variants of ICM_*x*PP.

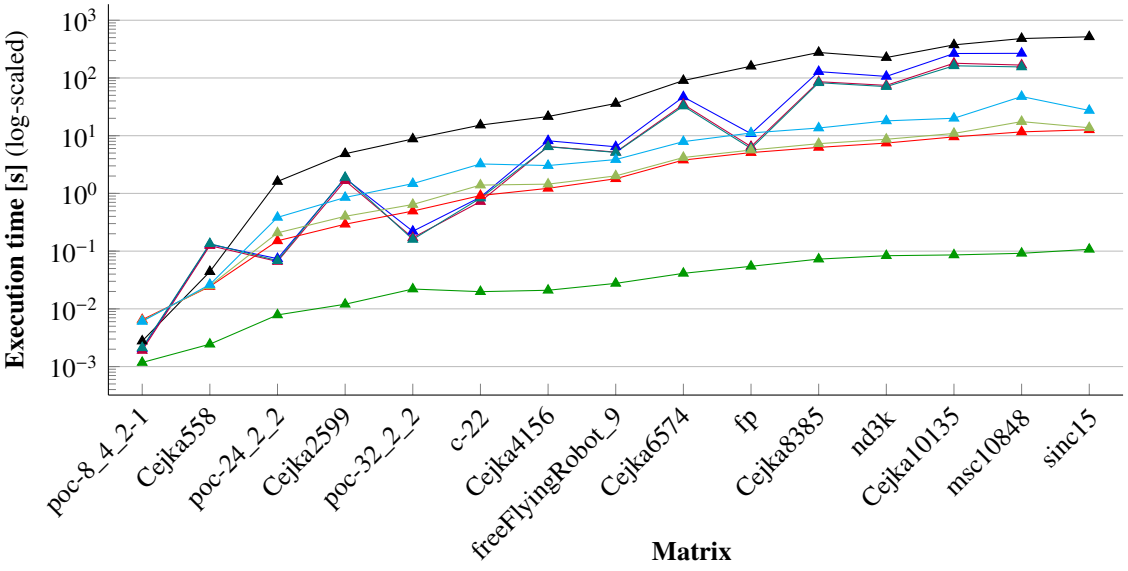
As Figure 3.2a shows, there are no zeros in the entire matrix except for some elements on the main diagonal which are not visible. Furthermore, since the matrix values were randomly generated, there were no patterns that would enable ICM_32PP to decompose the matrix rapidly. This is also apparent from Figure 3.2b, where it can be seen that the top-left and center sections were each computed in roughly 527 iterations.

In general, the number of iterations required to process a section of a *Cejka* dense matrix depends on the type of the section. In the case of a *diagonal section*, the number of iterations can be estimated as $2s + p$, where s represents the size of one dimension of a section and p denotes the number of elements pivoted in the section. This estimate comes from the observation that the number of iterations depends on the number of diagonal elements in a section. Specifically, the number of iterations performed per diagonal element varies based on two cases.

The first case assumes that the diagonal element is the bottom-right-most element of the section, and in this case, two iterations are performed. In the first iteration, the value of the diagonal element is computed. In the second iteration, the value is recomputed to assure that it has not changed. Note



(a) Double precision

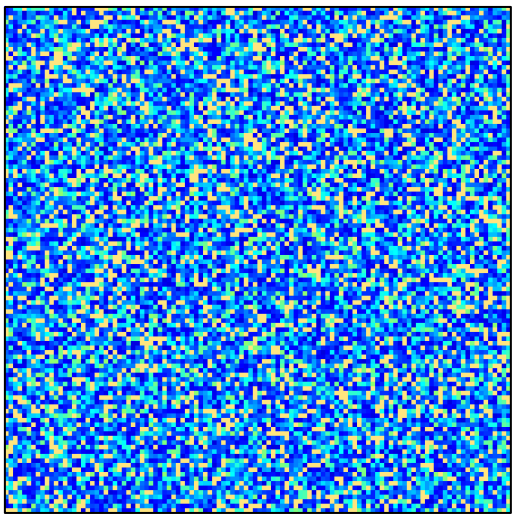


(b) Single precision

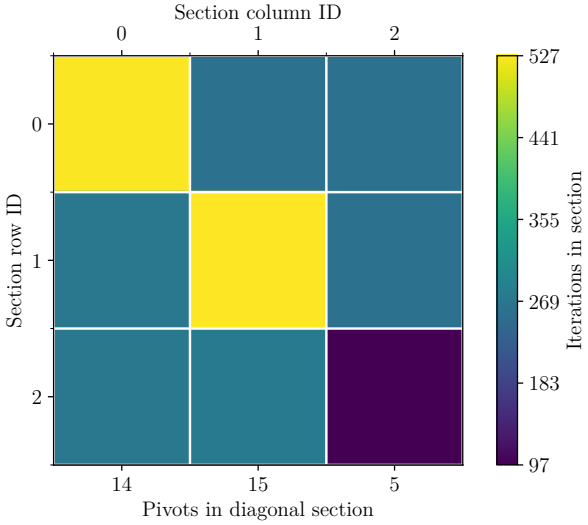
Figure 3.1: Execution times of decomposers listed in Table 3.3 on the subset of matrices shown in Table 3.2 using double and single precision. The vertical axis, representing execution time in seconds, is log-scaled for improved visibility of differences between implementations. Note that in the graph showing results for single precision, the data for the *sinc15* matrix is missing for the ICM_*x*PP decomposer as it failed to decompose the matrix.

that the two iterations are performed regardless of whether the element requires pivoting or not. If it does require pivoting, then the next steps involve processing the lower sections and pivoting the element.

The second case assumes the opposite, i.e., that the diagonal element is not the bottom-right-most element of the diagonal section. In this case, either two or three iterations are performed:



(a) Nonzero element pattern of the *Cejka558* matrix. The plot was generated using a Matlab script that utilizes the CSPY function from *SuiteSparse: A Suite of Sparse matrix packages at http://suitesparse.com* [54, 55]. In the plot, the color of nonzero elements depends on their absolute value. Small entries are light orange, large entries are black, and the color of mid-range entries ranges from light green to deep blue depending on the median of $\log_{10}$ of the nonzero values (with slight alterations) [55]. The color of zero entries is white.



(b) Iterative and pivoting metrics of ICM_32PP collected during the decomposition of the *Cejka558* matrix. Each square represents a 256-by-256 section of the matrix. The color in each square represents the number of iterations performed on that section. The left and top axes show the section ID in each dimension, while the bottom axis shows the number of pivots performed in a diagonal section. For example, the number of elements pivoted in section (0,0) was 14, while in section (1,1) it was 15. The plot was generated using a Python script.

Figure 3.2: Nonzero element pattern of the *Cejka558* matrix along with the iterative and pivoting metrics of ICM_32PP. The scripts used to generate the plots are available on request or as an attachment to this thesis.

- *Two* iterations are performed if the diagonal element does not require pivoting. In the first iteration, the diagonal element and the elements below it are computed. In the second iteration, the elements to the right of the diagonal element are computed.
- *Three* iterations are performed if the diagonal element requires pivoting. In such a case, the following occurs:

  1. In the first iteration, the diagonal element and the values below it are computed.
  2. Then, the iteration is performed again to assure that the element still requires pivoting and that the values below it are processed, i.e., they can be used for pivoting.
  3. The lower sections are then processed and the diagonal element is pivoted.
  4. In the third iteration, the elements to the right of the diagonal element are computed.

Additionally, if the diagonal section is the top-left section of the matrix, then the estimate is adjusted to $2s + p - 1$ since the first column of `LU` is the same as `A`. Therefore, in the first iteration, the values in the first column are already processed, which leads to the values to the right of the first diagonal element being finalized in the first iteration. Thus, an iteration is saved, hence the adjustment.

Note that in the context of this explanation, the computing of elements only refers to the computation of elements within the diagonal section.

According to the adjusted formula proposed above, the number of iterations required to process the top-left diagonal section of the *Cejka558* matrix is estimated to equal $2 \cdot 256 + 14 - 1 = 525$, which corresponds to the number of iterations performed. Similarly, according to the formula presented earlier ($2s + p$), the number of iterations required to process the center section was 527, which corresponds to the number of iterations performed.
In the case of the bottom-right diagonal section, the formula must be offset to account for the matrix bounds. For example, when considering the 558-by-558 matrix *Cejka558*, since each section represents a 256-by-256 block of elements, the bottom-right diagonal section would extend beyond the bounds of the matrix. Therefore, its boundaries are adjusted to match the boundaries of the matrix, resulting in a 46-by-46 block of elements. The processing of this section is expected to require $2 \cdot 46 + 5 = 97$ iterations. As shown in Figure 3.2b, this estimate is correct.

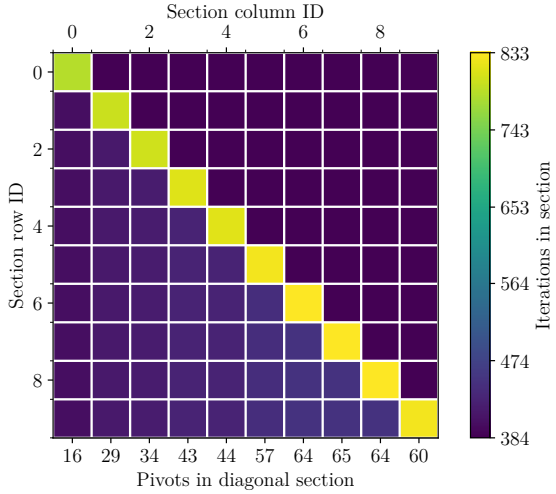
Continuing the context of the *Cejka* dense matrices, the number of iterations required to process a *lower section* is estimated to equal $s + p$, where s represents the size of one dimension of a section and p denotes the number of elements pivoted in the diagonal section above the lower section. Specifically, the number of iterations corresponds to the number of columns the section encompasses and the number of elements pivoted in the diagonal section above as described below:

- *Number of columns* - This is due to the chain of dependencies of each element. In other words, for a value in the lower section to be computed, it depends on the values to its left and the values in its column above the main diagonal. Since the latter are processed before the computation of the lower section begins, only the values to the left of the element remain to be processed. In the first iteration on a lower section, the values in the first column are processed. To clarify, in the first iteration, multiple columns of values in the section are computed, however, only the values in the first column are processed, i.e., their values are finalized in the first iteration. In the second iteration, the values in the second column are processed as they depend on the values processed in the first iteration. In summary, with each iteration, one column is processed.
- *Number of elements pivoted* - The addition of this value stems from the fact that before an element of a diagonal section is pivoted, the sections below the diagonal section are processed until, and including, the column containing the bad element. Thus, for every pivoted element, an extra iteration is performed to verify that the values up to and including the column are not changing, i.e., they are processed.

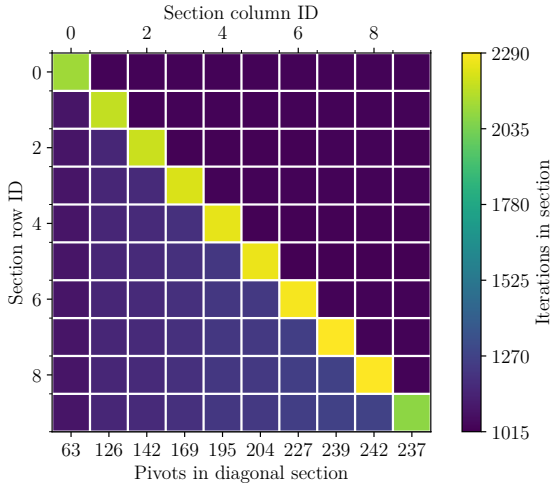
Note that even if the lower section is adjusted to fit the boundaries of the matrix, the number of iterations remains the same as the number of columns is not changed. For example, when considering the *Cejka558* matrix, the sections below the top-left diagonal section were processed in 270 iterations, which aligns with the prediction mentioned earlier: 256 + 14. Similarly, the section below the center diagonal section was processed in 272 iterations, while the prediction was 271.

The prediction of the number of iterations required to process a *right section* is similar to that of a lower section, except for the pivoting aspect. In other words, the number of iterations required to process a right section is estimated to equal $s + 1$, where s represents the number of rows the section encompasses. The additional iteration is performed to assure that the values of the right section are not changing, i.e., that they are processed. Similarly to the lower sections, the number of iterations required to process a right section is not affected if the section is adjusted to fit the boundaries of the matrix. For example, in the context of the *Cejka558* matrix, all right sections were processed in 257 iterations, which aligns with the prediction.

To show that the prediction is valid for other *Cejka* matrices, the iterative and pivoting metrics for ICM_32PP are presented in Figure 3.3 for matrices *Cejka3839* and *Cejka10135*.



(a) Iterative and pivoting metrics of ICM_32PP collected during the decomposition of the *Cejka3839* matrix. Each square represents a 384-by-384 section of the matrix.



(b) Iterative and pivoting metrics of ICM_32PP collected during the decomposition of the *Cejka10135* matrix. Each square represents a 1024-by-1024 section of the matrix.

Figure 3.3: Iterative and pivoting metrics of ICM_32PP collected during the decomposition of the *Cejka3839* and *Cejka10135* matrices.

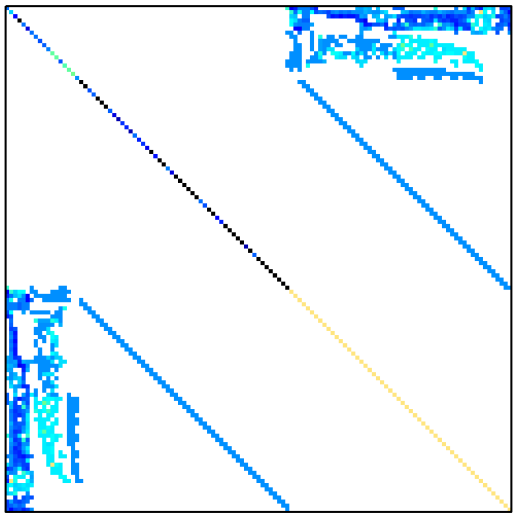
For example, in the case of *Cejka3839*, each right section took between 384 and 385 iterations to process, whereas the right sections of the *Cejka10135* matrix were processed in 1024 to 1025 iterations. The prediction for the former yields $384 + 1 = 385$ and the prediction for the latter yields $1024 + 1 = 1025$. Note that the difference in one iteration is most likely caused by the values of two columns being finalized in one iteration due to a favorable pattern. In Figures 3.3a and 3.3b, it can be observed that the colors of the diagonal and lower sections become lighter as the number of elements pivoted in each diagonal section increases. The only exception to this color lightening is the bottom-right-most diagonal section of the *Cejka10135* matrix. Similarly to the case with the last diagonal section of the *Cejka558* matrix, the section was adjusted to match the boundaries of the matrix. Therefore, instead of being a 1024-by-1024 section, it became a 919-by-919 section, requiring $2 \cdot 919 + 237 = 2075$ iterations to process.

Dense matrices with randomly generated values offer practically negligible opportunities for ICM_*x*PP to skip the computation of elements. Therefore, such matrices represent the worst-case scenario for the ICM_*x*PP decomposer, and the formulas derived from their decompositions can be considered as an inclusive upper bound estimate of the number of iterations required to process a section. In summary, for dense matrices with randomly-generated values, ICM_*x*PP effectively becomes an inefficient adaption of PCM_*x*PP.

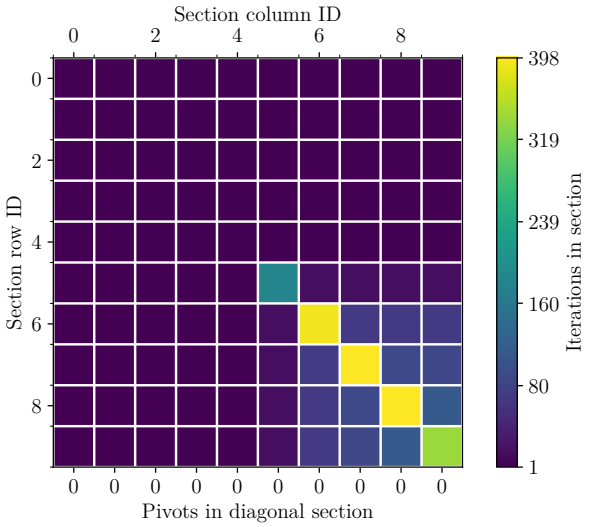
Steering away from instances of poor performance by ICM_*x*PP, there were three notable matrices out of the subset of 15 where ICM_*x*PP outperformed PCM_*x*PP using both double and single precision: *poc-24_2_2*, *poc-32_2_2*, and *c-22*. In general, it can be argued that ICM_*x*PP outperformed PCM_*x*PP on the aforementioned matrices, as they required minimal, if any, pivoting. Seeing as the matrices prefixed with *poc* were obtained from the "Poisson equation on a cube" problem implemented in BDDCML, they are discussed as part of Section 3.2.

When using double precision, the best-performing variant of ICM_*x*PP, ICM_32PP, decomposed the *c-22* matrix in 1.313 seconds, while the best-performing variant of PCM_*x*PP, PCM_8PP, decomposed it in 2.192 seconds. When single precision was used, ICM_32PP decomposed the matrix in 0.830 seconds, while PCM_8PP decomposed it in 0.917 seconds. The main reason why

ICM_*x*PP performed well specifically on the *c-22* matrix is that the initial estimate, `A`, allowed for the majority of the sections to be processed in one iteration, as shown in Figure 3.4. To provide context for the iterative metrics, the nonzero element pattern of `LU` is presented in Figure 3.5.



(a) Nonzero element pattern of the *c-22* matrix.



(b) Iterative and pivoting metrics of ICM_32PP collected during the decomposition of the *c-22* matrix. Each square represents a 384-by-384 section of the matrix.

Figure 3.4: Nonzero element pattern of the *c-22* matrix along with the iterative and pivoting metrics of ICM_32PP. In the metrics plot, the color in each square represents the number of iterations performed on that section. The left and top axes show the section ID in each dimension, while the bottom axis shows the number of pivots performed in a diagonal section. However, since no elements were pivoted, the values are all zero.

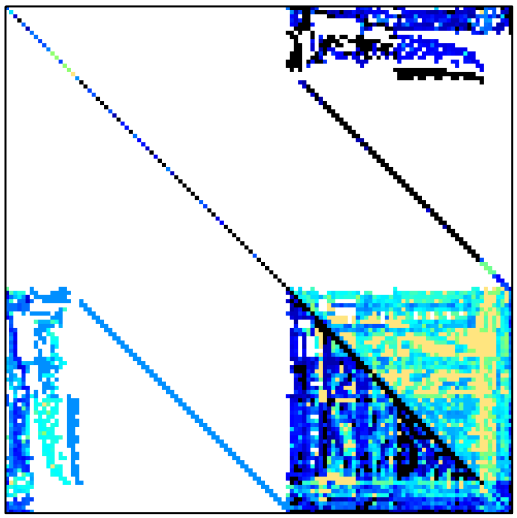


Figure 3.5: Nonzero element pattern of the `LU` matrix produced by ICM_32PP through the decomposition of the *c-22* matrix.

As Figure 3.4a shows, the main diagonal comprises nonzero elements. Therefore, the conditional partial pivoting of ICM_*x*PP was not required. Furthermore, according to the iterative metrics in Figure 3.4b and the resulting `LU` matrix shown in Figure 3.5, it can be seen that all sections until section (5, 5) took only one iteration to process. Thus, as mentioned before, the initial estimate

of `LU` in the form of `A`, described in Section 2.2.2, was instrumental in the efficient decomposition of the *c-22* matrix. In this case, the initial estimate was suitable since the input matrix mainly had elements on the main diagonal. Therefore, ICM_*x*PP used the elements of the main diagonal of `A` in the main diagonal of `L`. Then, the multiplication of `L` and `U`, which has a unit diagonal, essentially meant that `L` was multiplied by a matrix resembling the identity matrix. As the figures further show, the elements in the bottom-left and top-right of `LU` disrupted the simple computation of the main diagonal. Subsequently, the bottom-right part of `LU` required many iterations to account for the disturbance. The claim that ICM_*x*PP performs well on matrices with most of their nonzero elements on the main diagonal corresponds to the findings presented in Chapter 3 of *Parallel LU Decomposition for the GPU* [2].

Next, the variants of ICM_*x*PP will be compared to each other. While Figure 3.1 presents the execution times of all variants of ICM_*x*PP, it may be difficult to discern the differences in performance. Thus, to compare the performance of the variants of ICM_*x*PP, Figure 3.6 shows their execution times on the subset of matrices when double precision was used. Note that unlike in Figure 3.1, the vertical axis in Figure 3.6 is not log-scaled to better portray the differences in performance among the variants.

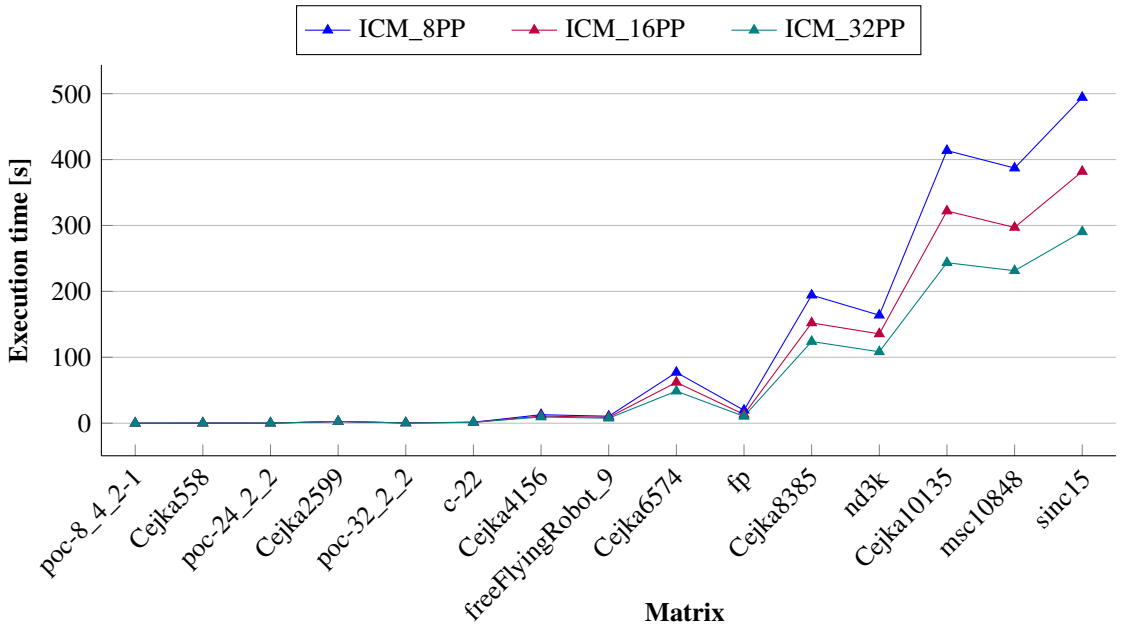


Figure 3.6: Execution times of the variants of ICM_*x*PP on the subset of matrices shown in Table 3.2 using double precision.

Out of the three variants of ICM_*x*PP that were benchmarked on the subset of 15 matrices, ICM_32PP performed the best and ICM_8PP the worst, as can be seen in Figure 3.6.

To determine the reason behind the differences in performance between the variants, it is necessary to consider different aspects of the implementations. Since the section size is near-identical for all variants, and the difference in the number of threads performing a row swap in each variant does not affect the performance, the only remaining major factor is the kernels computing the iterations. Disregarding the changes in the implementations of the kernels since they were benchmarked in *Parallel LU Decomposition for the GPU* [2], it can be argued that the difference in the performance of the variants of ICM_*x*PP is due to uncoalesced and inefficient global memory access.

First, the issue of uncoalesced global memory access will be described. In the context of the Decomposition benchmark for decomposers, the matrices were stored in row-major order on the

GPU for the ICM_*x*PP decomposer. As a result, uncoalesced access to global memory occurs for ICM_8PP and ICM_16PP.

For example, in the case of ICM_8PP, this is due to it using 8-by-8 thread blocks. Since threads consecutive in their 1st dimension are grouped into warps of 32 threads, each block of threads in ICM_8PP is made up of two warps. The first warp consists of threads in the first four rows of the thread block, while the remaining threads belong to the second warp. When these threads read matrix elements from global memory, the first row of eight threads reads neighboring elements, and the next row of threads also reads neighboring elements. However, the second group of eight elements is not adjacent to the first group of eight elements. In general, while each group of eight threads within a warp accesses neighboring global memory addresses, the accessed elements of each group are not adjacent to those accessed by the other groups of eight threads. Therefore, instead of the threads within a warp all reading neighboring elements in parallel, i.e., in one read transaction, they perform $32/8 = 4$ sequential read transactions [2].

Similarly, since ICM_16PP uses 16-by-16 thread blocks, each warp performs $32/16 = 2$ sequential transactions when reading from global memory during kernel execution. Conversely, this issue is not present for ICM_32PP since each row of a thread block comprises one warp of threads. Thus, the threads of each warp read 32 neighboring elements in one transaction.

The next issue, inefficient global memory access, describes a scenario where the smaller the thread blocks are, the more accesses to global memory they perform. In Section 2.2.2 the following was mentioned in terms of the computation kernels of ICM_*x*PP:

> *... 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 from global memory to shared memory, 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.*

When a block of threads performs matrix multiplication by iterating over blocks of elements, each iteration involves a 32-by-32 block of threads reading from global memory only once. On the other hand, to cover the same block of elements, 16 8-by-8 blocks of threads are required. Specifically, four 8-by-8 blocks of threads stacked on top of each other are required to compute eight columns, and there are four parts consisting of eight columns each in 32 columns. Therefore, in each iteration, the four blocks stacked on top of each other will read the same 8-by-8 block of elements four times in total, since they cannot access the shared memory of the other thread blocks. This means that when a 32-by-32 block of threads computes the values of a 32-by-32 block of elements in `LU`, it performs fewer read operations from global memory compared to using multiple 8-by-8 blocks of threads to compute the same block of elements.

The consequences of these inefficiencies are visible in Figure 3.6, where, especially for matrices that take a long time to decompose, ICM_32PP consistently outperforms ICM_16PP, which in turn consistently outperforms ICM_8PP.

CMPP As can be seen from Figure 3.1, the CMPP decomposer performed worse than the other benchmarked decomposers on all matrices of the subset, except for the *poc-8_4_2-1* and *Cejka558* matrices. The lack of performance can be attributed to two main reasons: the algorithm of CMPP is strictly sequential, and its implementation is not optimized. The performance of CMPP greatly depends on the dimensions of the input matrix, therefore, only the execution times it achieved on the

smallest and largest matrices of the subset will be mentioned. When using double precision, CMPP decomposed the *poc-8_4_2-1* matrix in 0.003 seconds and the *sinc15* matrix in 536.824 seconds. When using single precision, CMPP decomposed the matrices in 0.003 and 517.045 seconds, respectively. For context, the slowest variant of ICM_*x*PP, ICM_8PP, decomposed the *sinc15* matrix in 494.174 seconds using double precision. Ultimately, in the context of the Decomposition benchmark for decomposers, CMPP served as a baseline against which other decomposers were expected to perform better. Additionally, it served as a means to verify the accuracy of results produced by the other decomposers.

PCM_*x*PP Overall, on the subset of 15 matrices, PCM_*x*PP was the second fastest decomposer. Since its algorithm is a parallelized version of CMPP's algorithm, its performance also heavily depends on the dimensions of the input matrix, as shown in Figure 3.1. For the analysis of PCM_*x*PP, Figure 3.7 displays the execution times of its variants when double precision was used.

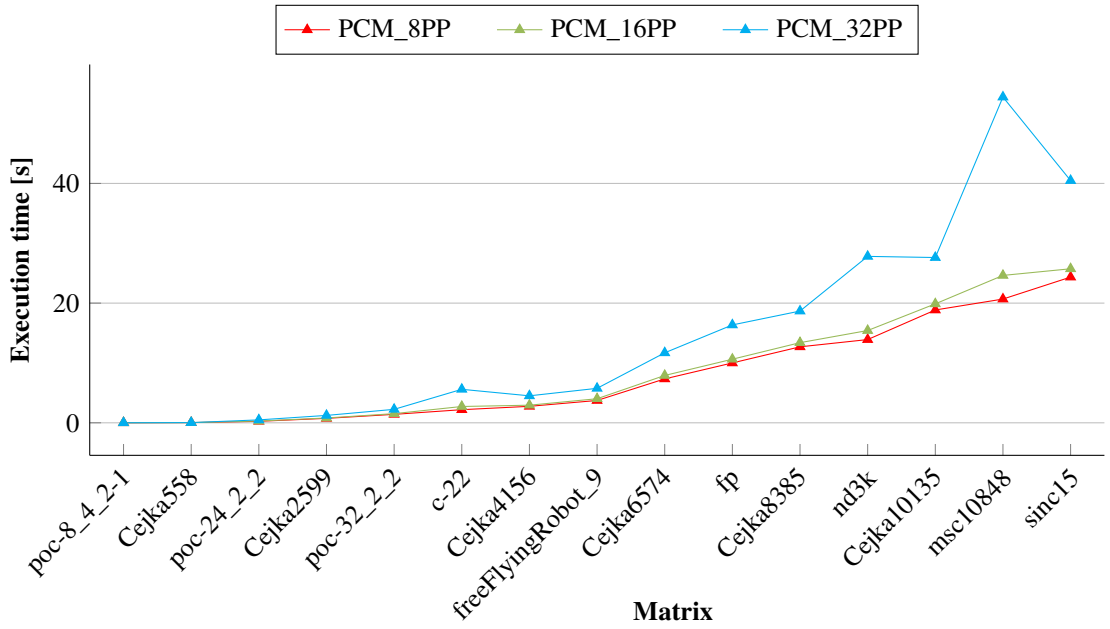


Figure 3.7: Execution times of the variants of PCM_*x*PP on the subset of matrices shown in Table 3.2 using double precision.

As can be seen in Figure 3.7, PCM_8PP consistently outperforms PCM_16PP, which in turn consistently outperforms PCM_32PP. Similarly to ICM_*x*PP, the difference in execution times increases with larger matrix dimensions.

In the context of the Decomposition benchmark for decomposers, the input matrix was stored in row-major order, which resulted in slightly lower execution times for the variants compared to when column-major ordering was used. The higher performance of the decomposer with row-major ordering was unexpected since it meant that accesses to global memory were not coalesced. As mentioned in Section 2.2.2, the thread blocks of PCM_*x*PP are one-dimensional and consist of x^2 threads. In the context of PCM_32PP, when the threads of a block read a column of elements from global memory in each iteration of the for loop shown in Listing B.2, the addresses accessed by the threads are not adjacent. As a result, the global memory access of threads belonging to each warp is not coalesced. This means that the read operations performed by the warp are divided into more memory transactions compared to coalesced global memory access. Nevertheless, uncoalesced access to global memory did not seem to have a noticeable impact on the performance of PCM_*x*PP.

The differences in the performance of the variants of PCM_*x*PP can be attributed to idle resources. This claim is based on the observation that the execution time of PCM_32PP and PCM_16PP consistently appears to be worse than that of PCM_8PP for larger sparse matrices. The GPU used for the benchmarks, Nvidia A100, has the following limitations per SM [9]:

1. maximum threads per SM: 2048,
2. maximum warps per SM: 64, and
3. maximum thread blocks per SM: 32.

Therefore, the PCM_32PP decomposer, which uses blocks of 1024 threads, can only schedule two of its blocks per SM before the threads-per-SM limit is reached. Moreover, a block of threads cannot be released from an SM until all of its warps have completed their work [56]. Consequently, if one warp within the block takes longer to complete its work, it can potentially hold the entire block of 1024 threads in the SM, preventing the scheduling of another thread block. This situation results in idle resources on the SM, as the other warps have already completed their work. Notably, this scenario is more likely to occur with blocks consisting of a larger number of threads.

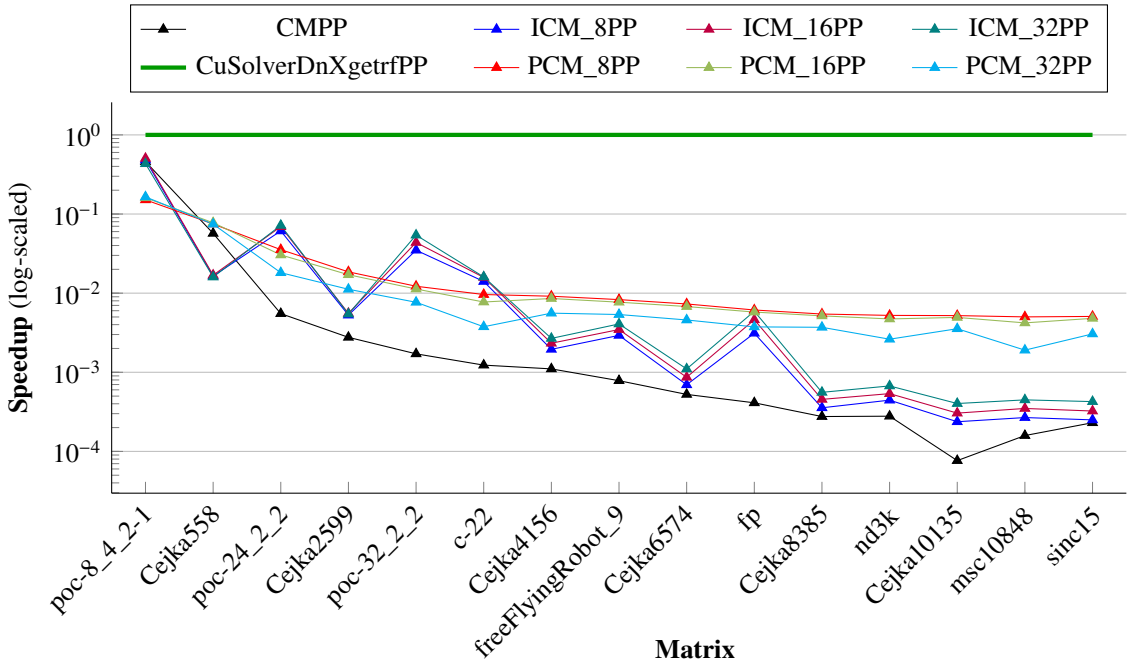
This congestion is less likely to occur for blocks with fewer threads since each SM can handle more blocks simultaneously. For example, PCM_16PP, which uses blocks of 256 threads, can schedule up to eight of its blocks per SM. Similarly, PCM_8PP can schedule up to 32 of its blocks, each consisting of 64 threads, per SM.

Earlier, it was mentioned that this behavior can be observed for larger sparse matrices. This is due to larger matrices generally requiring more computing threads, thus necessitating more thread blocks. Furthermore, the sparsity of the matrices increases the likelihood of an unequal distribution of work across warps, as the majority of elements are zero. In other words, during the decomposition of sparse matrices, it is more likely for many warps to be working with zeros while other warps are working with nonzeros, which results in the latter requiring more time and causing the idleness of resources.

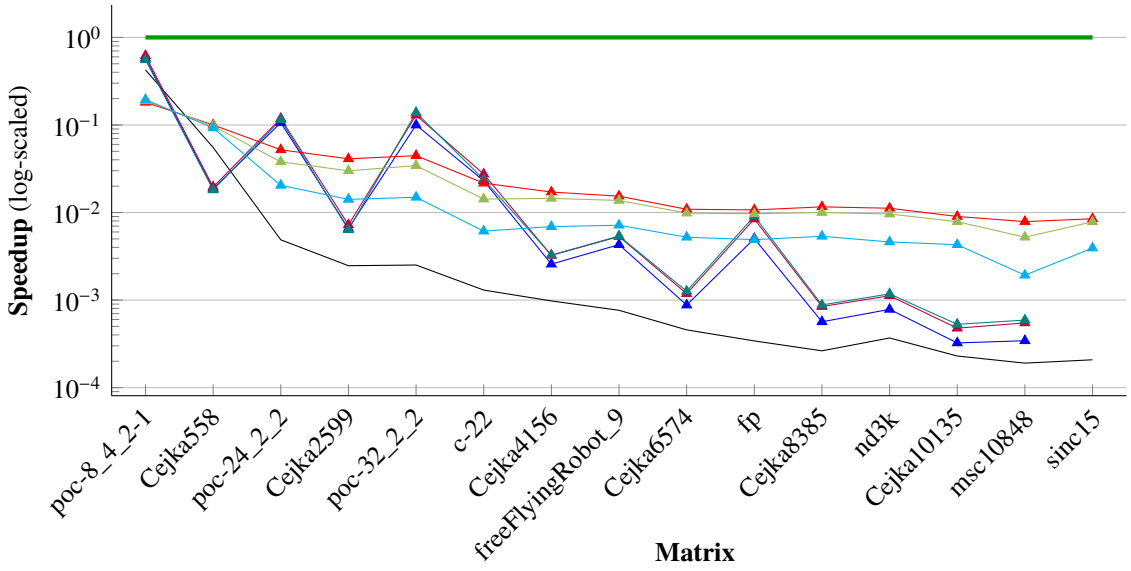
This behavior seems to occur in Figure 3.7, where the execution time of PCM_32PP appears to spike for larger sparse matrices such as *nd3k* and *msc10848*. Additionally, the figure shows that the PCM_16PP decomposer also exhibits a noticeable spike in execution time for the *msc10848* sparse matrix. In contrast, the PCM_8PP decomposer, which employs smaller thread blocks compared to the other variants, shows the best performance.

It should be noted once again that the implementation of PCM_*x*PP was not optimized in any way. This means that optimization techniques such as shared memory usage, block-level loop unrolling, and others were not employed. Consequently, as mentioned in Section 2.2.2, PCM_*x*PP is highly inefficient and fails to fully exploit the computational power of the GPU. However, despite its lack of optimization, PCM_*x*PP still demonstrates better performance compared to the complex implementation of ICM_*x*PP. This suggests that ICM_*x*PP may be suitable for specific types of matrices, such as those encountered in the BDDCML benchmark discussed in Section 3.2.

Performance Comparison Relative to CuSolverDnXgetrfPP For completeness, this section presents the performance comparison of the decomposers listed in Table 3.3 relative to the CuSolverDnXgetrfPP decomposer. First, the speedup relative to the CuSolverDnXgetrfPP decomposer, using both double and single precision, is presented in Figure 3.8. Then, the execution times of the decomposers for both double and single precision are presented in Tables 3.4 and 3.5, respectively. Note that the tables only show the execution times of the best-performing variants of ICM_*x*PP and PCM_*x*PP.



(a) Double precision



(b) Single precision

Figure 3.8: Speedup comparison of the decomposers listed in Table 3.3 relative to CuSolverDnXgetrfPP on the subset of matrices shown in Table 3.2 using double and single precision. The vertical axis is log-scaled for improved visibility of differences between implementations.

Figure 3.8, and Tables 3.4 and 3.5 suggest that CuSolverDnXgetrfPP is the fastest decomposer among those benchmarked in the Decomposition benchmark for decomposers. However, the results presented only compare the decomposers on a subset of 15 matrices. To provide further support for the claim, it is necessary to evaluate the performance of CuSolverDnXgetrfPP on the entire set of 50 matrices.

Matrix	CuSolverDnXgetrfPP	CMPP	ICM_32PP	PCM_8PP
poc-8_4_2-1	0.0012	0.0026	0.0028	0.0079
Cejka558	0.0025	0.0431	0.1536	0.0328
poc-24_2_2	0.0089	1.6072	0.1233	0.2498
Cejka2599	0.0138	4.9971	2.5359	0.7432
poc-32_2_2	0.0172	10.0834	0.3185	1.4109
c-22	0.0211	17.0976	1.3125	2.1916
Cejka4156	0.0252	22.8199	9.3951	2.7534
freeFlyingRobot_9	0.0309	39.4910	7.5774	3.7317
Cejka6574	0.0536	102.0215	48.6333	7.3379
fp	0.0613	149.0108	10.3767	10.0030
Cejka8385	0.0692	250.6601	123.9475	12.6962
nd3k	0.0728	261.3578	108.3278	13.9076
Cejka10135	0.0981	1282.0280	243.4973	18.8531
msc10848	0.1037	653.4609	231.3460	20.6770
sinc15	0.1237	536.8241	290.3724	24.3491

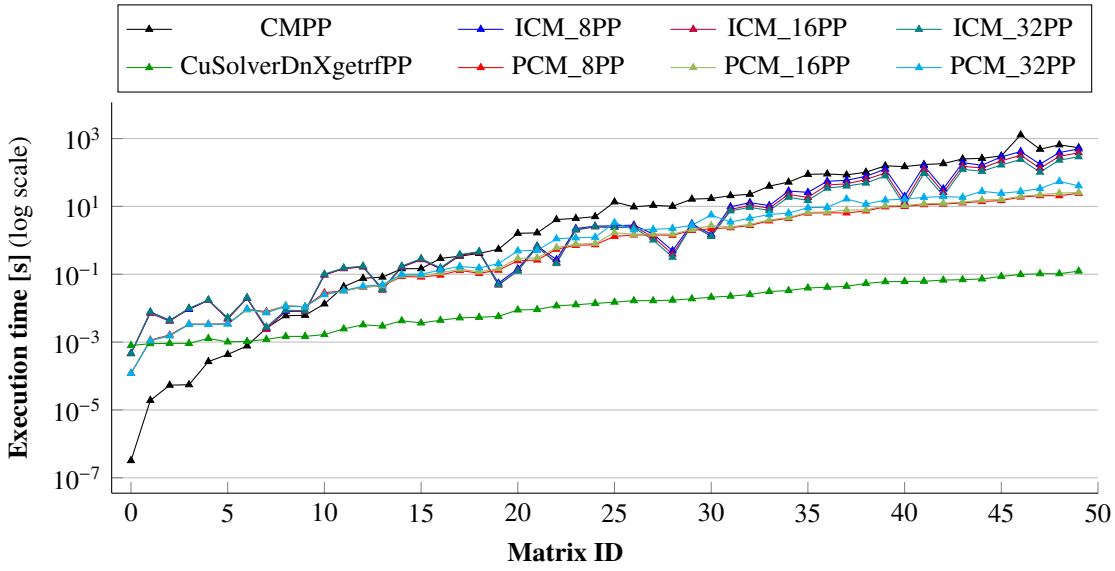
Table 3.4: Execution times (in seconds) of the decomposers listed in Table 3.3 on the set of matrices (Table 3.2) using *double* precision. Only the execution times of the best-performing variants of ICM_*x*PP and PCM_*x*PP are presented.

Matrix	CuSolverDnXgetrfPP	CMPP	ICM_32PP	PCM_8PP
poc-8_4_2-1	0.0012	0.0028	0.0021	0.0065
Cejka558	0.0024	0.0441	0.1347	0.0244
poc-24_2_2	0.0079	1.6099	0.0684	0.1509
Cejka2599	0.0120	4.8680	1.8898	0.2926
poc-32_2_2	0.0221	8.7752	0.1594	0.4945
c-22	0.0199	15.2680	0.8297	0.9173
Cejka4156	0.0210	21.4828	6.4306	1.2250
freeFlyingRobot_9	0.0276	36.1487	5.1294	1.7961
Cejka6574	0.0412	90.2590	32.6642	3.7836
fp	0.0546	159.8940	6.0036	5.0968
Cejka8385	0.0727	276.4194	82.7072	6.2676
nd3k	0.0834	225.9154	70.8304	7.4513
Cejka10135	0.0859	374.2370	162.5578	9.5356
msc10848	0.0917	481.6722	155.3406	11.6422
sinc15	0.1075	517.0449	-	12.6665

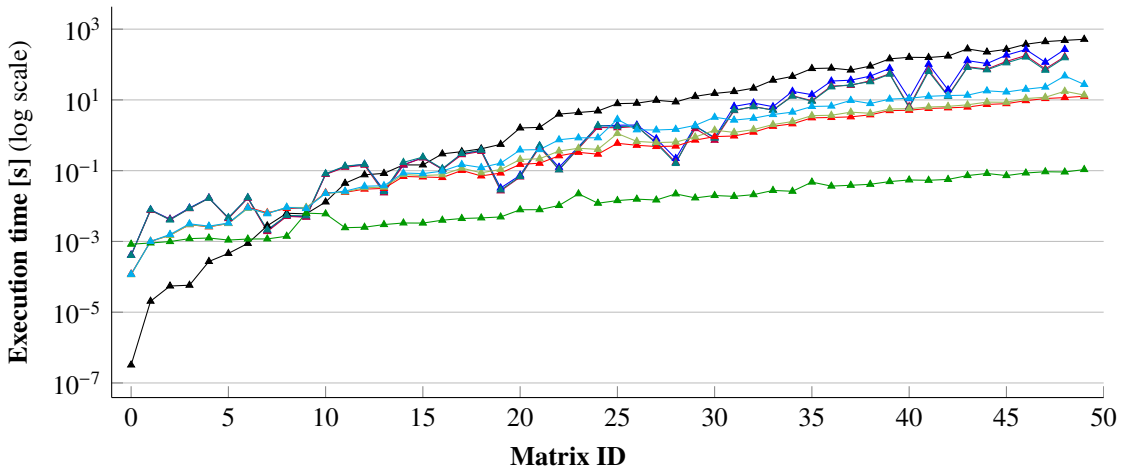
Table 3.5: Execution times (in seconds) of the decomposers listed in Table 3.3 on the set of matrices (Table 3.2) using *single* precision. Only the execution times of the best-performing variants of ICM_*x*PP and PCM_*x*PP are presented. The execution time of ICM_32PP is missing for the *sinc15* matrix as ICM_*x*PP failed to decompose the matrix using single precision.

Comparison of Execution Times on All Matrices

This section presents a comparison of the execution times of the decomposers listed in Table 3.3 on the set of 50 matrices using both double and single precision. The benchmarks were run on the platform specified in Section 3.1.1. Similarly to the benchmark results presented in Section 3.1.3, the matrices were sorted according to their dimensions from smallest to largest. The execution times are presented in Figure 3.9.



(a) Double precision



(b) Single precision

Figure 3.9: Execution times of decomposers listed in Table 3.3 on the entire set of 50 matrices using double and single precision. The vertical axis, representing execution time in seconds, is log-scaled for improved visibility of differences between implementations. Note that in the graph depicting the results for single precision, the data for the *sinc15* matrix is not available for the ICM_*x*PP decomposer as it failed to decompose the matrix.

As depicted in Figure 3.9, CuSolverDnXgetrfPP exhibits the best performance among the benchmarked decomposers. The only scenario where it was surpassed by all other decomposers was with a 4-by-4 testing matrix. Additionally, for matrices with IDs 0 to 7, it exhibited slower performance compared to CMPP. However, it is worth noting that the dimensions of matrices 0 to 7 range from 4-by-4 to 230-by-230; such matrices are not commonly encountered in real-life problems.

In general, PCM_*x*PP and ICM_*x*PP exhibited faster performance when using single precision compared to double precision. On the other hand, CuSolverDnXgetrfPP demonstrated only minor differences in performance between double and single precision.

The figures provide additional support for the claim stated in the analysis of ICM_*x*PP in Section 3.1.3 that larger thread blocks have a positive impact on its performance. For instance,

Figure 3.9a clearly illustrates the difference in performance between the variants of ICM_*x*PP. Specifically, ICM_32PP consistently outperforms ICM_16PP, which in turn consistently outperforms ICM_8PP.

Similarly, the figures provide additional support for the claim made in the analysis of PCM_*x*PP in Section 3.1.3 that smaller thread blocks decrease resource idleness. Specifically, PCM_8PP consistently outperforms PCM_16PP, which in turn consistently outperforms PCM_32PP. For example, when considering double precision, PCM_32PP exhibited a significant spike in execution time for the sparse matrices listed in Table 3.6.

ID	Matrix	Rows/Columns	Nonzeros	Avg. nonzeros per row
25	bayer06.mtx	3008	20715	6.887
30	c-22.mtx	3792	28870	7.613
44	nd3k.mtx	9000	3279690	364.410
48	msc10848.mtx	10848	1229776	113.364

Table 3.6: Sparse matrices for which the execution time of PCM_32PP spiked compared to the other variants of PCM_*x*PP. The average number of nonzeros per row is rounded to three decimal places.

Note that, in Figure 3.9, the spikes in execution time of PCM_32PP are more pronounced for the larger matrices listed in Table 3.6. This was one of the key points of the claim made in the analysis of PCM_*x*PP. However, Figure 3.9a also shows a spike in the execution time of PCM_32PP for matrix 37 (*Cejka6192*), which is a dense matrix. Since this behavior does not occur for any of the other dense matrices, the reason for this sudden increase in execution time remains unclear.

To put the results of the entire set of matrices into perspective, Table 3.7 presents the total time taken by each decomposer to decompose the entire set of 50 matrices.

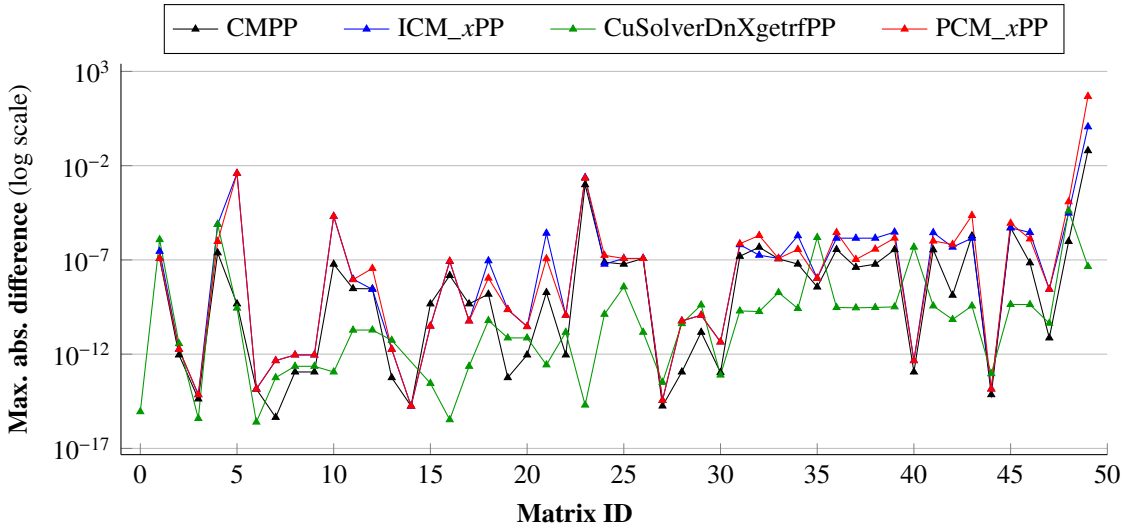
Decomposer	Total execution time [s]	
	Double precision	Single Precision
CuSolverDnXgetrfPP	1.41	1.30
CMPP	5034.38	3742.05
ICM_8PP	2751.50	1459.26
ICM_16PP	2138.09	984.42
ICM_32PP	1669.74	920.30
PCM_8PP	221.59	113.10
PCM_16PP	240.95	133.38
PCM_32PP	387.39	275.34

Table 3.7: Total execution time (in seconds) taken by each decomposer to decompose the set of 50 matrices on the RCI cluster, specified in Section 3.1.1, using double and single precision. The execution times are rounded to three decimal places.

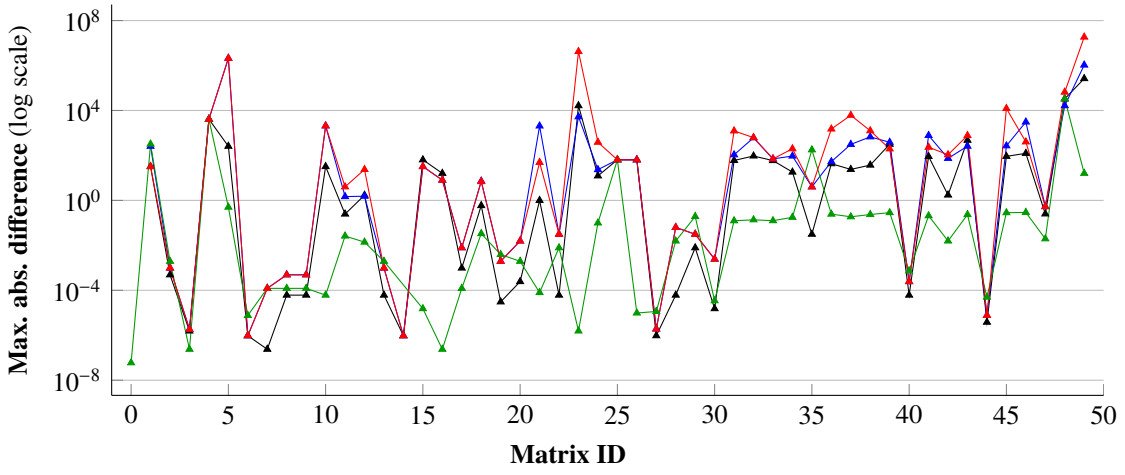
In terms of execution speed, both Figure 3.9 and Table 3.7 demonstrate that CuSolverDnXgetrfPP is the fastest decomposer. Excluding CuSolverDnXgetrfPP, the fastest decomposer is PCM_*x*PP, specifically the PCM_8PP variant. Although the variants of ICM_*x*PP were generally slower than the variants of PCM_*x*PP for most matrices, they exhibited promising performance for specific matrix types, such as the *poc* matrices obtained from the BDDCML benchmark.

Accuracy of Results on All Matrices

Another important aspect in the performance of a decomposer is the accuracy of the results it produces. The maximum absolute difference between the expected results and the actual results for the entire set of 50 matrices is shown in Figure 3.10. For context, the maximum absolute difference for decomposers was defined in Equation 2.7 as max $|\mathbf{A} - \mathbf{LUP}|$, where $\mathbf{A}$ is the input matrix, $\mathbf{L}$ and $\mathbf{U}$ are the results of the decomposition operation, and $\mathbf{P}$ is the permutation matrix. Note that the permutation matrix was represented by a vector in the implementations. This section mentions decomposers ICM_*x*PP and PCM_*x*PP instead of their variants, as the variations of the decomposers did not differ in terms of accuracy, i.e., there was no need to specifically address each variant.



(a) Double precision



(b) Single precision

Figure 3.10: Accuracy of results achieved by the decomposers listed in Table 3.3 (excluding the different variants) on the entire set of 50 matrices using both double and single precision. The matrix ID signifies the ID of the matrices after they have been sorted according to their dimension from smallest to largest. The vertical axis is log-scaled for improved visibility. Note that the accuracy values for the first matrix for CMPP, ICM_*x*PP, and PCM_*x*PP are not plotted as their results matched the expected results, i.e., the plotted values would be equal to $\log 0$. Additionally, it is important to mention that the vertical axes of both graphs do not cover the same range.

As depicted in Figure 3.10, the decomposers implemented by the author of this thesis exhibited an acceptable level of accuracy in comparison to the established CuSolverDnXgetrfPP when double precision was used. However, when it came to single precision, the accuracy of the decomposers was found to be inferior to that of CuSolverDnXgetrfPP. Given the large discrepancy in result accuracy between double and single precision, they will be analyzed separately.

Double Precision In addition to the accuracy of results shown in Figure 3.10a, a selection of statistical indices is presented in Table 3.8 to offer additional insights into the performance of the decomposers.

Decomposer	Mean	Std. dev.	Q1	Q2	Q3	Max.
CMPP	1.27e−3	8.84e−3	1.14e−13	2.40e−9	1.18e−7	0.063
CuSolverDnXgetrfPP	1.02e−6	5.74e−6	2.27e−13	4.27e−11	3.52e−10	4.01e−5
ICM_*x*PP	0.023	0.163	1.06e−11	7.12e−8	1.42e−6	1.152
PCM_*x*PP	0.948	6.705	1.06e−11	6.11e−8	8.96e−7	47.414

Table 3.8: Statistical indices of the accuracy of results achieved by the decomposers in the Decomposition benchmark using double precision. Columns Q1, Q2, and Q3 represent the first, second, and third quartiles, respectively. The indices were computed using LibreOffice Calc, a spreadsheet software.

The statistical indices in Table 3.8 show that the CuSolverDnXgetrfPP decomposer produced the most accurate results followed by CMPP. However, cases where the CuSolverDnXgetrfPP decomposer produced less accurate results than all remaining decomposers have been observed, e.g., for matrices 35 (*exdata_1*) and 40 (*fp*). For these matrices, the relative inaccuracy of CuSolverDnXgetrfPP is present for both double and single precision, as Figures 3.10a and 3.10b show, respectively. The reasons for the relative inaccuracy are unclear as there is no consistent pattern that would indicate a matrix characteristic unfavorable to CuSolverDnXgetrfPP.

For example, it was theorized that ICM_*x*PP produced more accurate results when decomposing matrices 35 and 40 due to their higher condition numbers, 1.06e+12 and 4.78e+12, respectively. In other words, the iterative approach of ICM_*x*PP was theorized to produce accurate results when decomposing numerically unstable matrices. However, this theory was disproved by a counterexample, as the ICM_*x*PP decomposer produced less accurate results than CuSolverDnXgetrfPP on matrix 26 (*bayer09*), which has a condition number equal to 2.63e+21. Note that the condition numbers were computed using a Matlab script that utilizes the approach presented in *Algorithm 907* [57, 58, 59]. The script is available on request or as an attachment to this thesis.

In the description of the implementation of the ICM_*x*PP decomposer in Section 2.2.2, it was mentioned that the processing tolerance was specifically chosen to be zero. The reasoning behind this claim was that setting the tolerance to even a small nonzero value, e.g., 1e−5, could potentially decrease the accuracy. To support this claim, the accuracy of results for various processing tolerances is presented in Figure 3.11.

As can be seen in Figure 3.11, ICM_*x*PP with its processing tolerance set to zero is the most accurate among the explored processing tolerances. It is noteworthy that ICM_*x*PP with its processing tolerance set to 1e−10 also produces accurate results. Interestingly, for four matrices, it produced marginally more accurate results compared to the zero-tolerance variant. The reason behind this behavior is unclear. Nevertheless, the accuracy of the 1e−10 variant was inferior to that of the zero-tolerance variant in multiple cases.
For the remaining processing tolerances, ICM_*x*PP produced less accurate results compared to

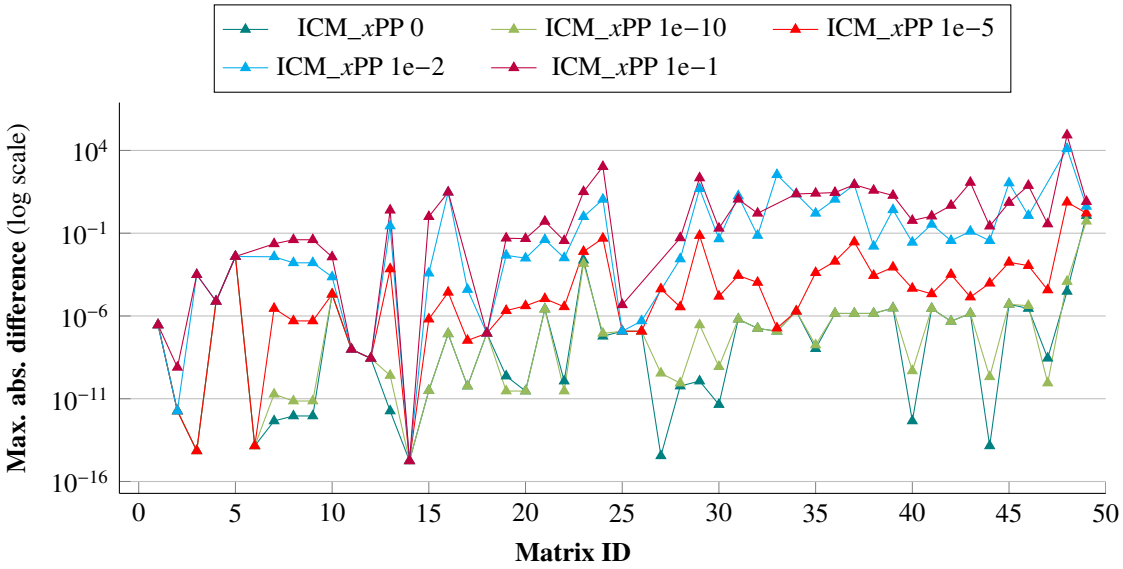


Figure 3.11: Accuracy of the results achieved by ICM_*x*PP with various processing tolerances on the entire set of 50 matrices using double precision. The name of each decomposer follows the naming pattern *decomposer <processing_tolernace>*. The vertical axis is log-scaled for improved visibility. Note that the accuracy values for the first matrix are not plotted as the decomposers produced results produced that matched the expected results, i.e., the plotted value would be equal to log 0. Additionally, certain accuracy values are missing as their decomposers failed to decompose those particular matrices. Specifically, ICM_*x*PP 1e−2 failed to decompose matrices 6 (*west0156*), 27 (*garon1*), and 47 (*TSOPF_FS_b162_c1*), while ICM_*x*PP 1e−1 failed to decompose matrices 6 (*west0156*), 17 (*b2_ss*), 27 (*garon1*), 33 (*freeFlyingRobot_9*), and 47 (*TSOPF_FS_b162_c1*).

the 1e−10 variant. However, the variants with looser processing tolerances may still be useful if ICM_*x*PP is employed as a preconditioner, particularly in scenarios where high accuracy may not be necessary. An example of such a scenario is the "Poisson equation on a cube" problem, whose benchmark results are discussed in Section 3.2.

With regard to the ICM_*x*PP decomposer with its processing tolerance set to zero, the obtained data seems to contradict the theory proposed in *Parallel LU Decomposition for the GPU* [2]: "*Since Crout's method is direct, it can be theorized that rounding errors may result in it providing less accurate results compared to its numerical modification*". However, it is important to note that the theory was formulated without considering partial pivoting. Therefore, the results of this benchmark indicate that combining the iterative approach of ICM*x* and partial pivoting is not a viable approach for LU decomposition in terms of accuracy.

When considering the CMPP and PCM_*x*PP decomposers, the values in Table 3.8 highlight occasional inaccuracies. For example, in Figure 3.10a, it can be seen that both decomposers exhibit large discrepancies between the expected and actual results for matrix 50 (*sinc15*). This inaccuracy is reflected in their mean values, as shown in Table 3.8, deviating significantly. Additionally, the relatively small value of the third quartile further supports this observation. The inaccuracy of CMPP, a naive and strictly sequential implementation, can be attributed to conditional partial pivoting as the accuracy of CMPP was higher when every element was pivoted regardless of its value. The comparison of the accuracy of results between CMPP and PCM_*x*PP, considering both conditional partial pivoting and *full partial pivoting*, is presented in Figure 3.12. The term *full partial pivoting* represents partial pivoting where every element of the main diagonal is pivoted, regardless of its value.

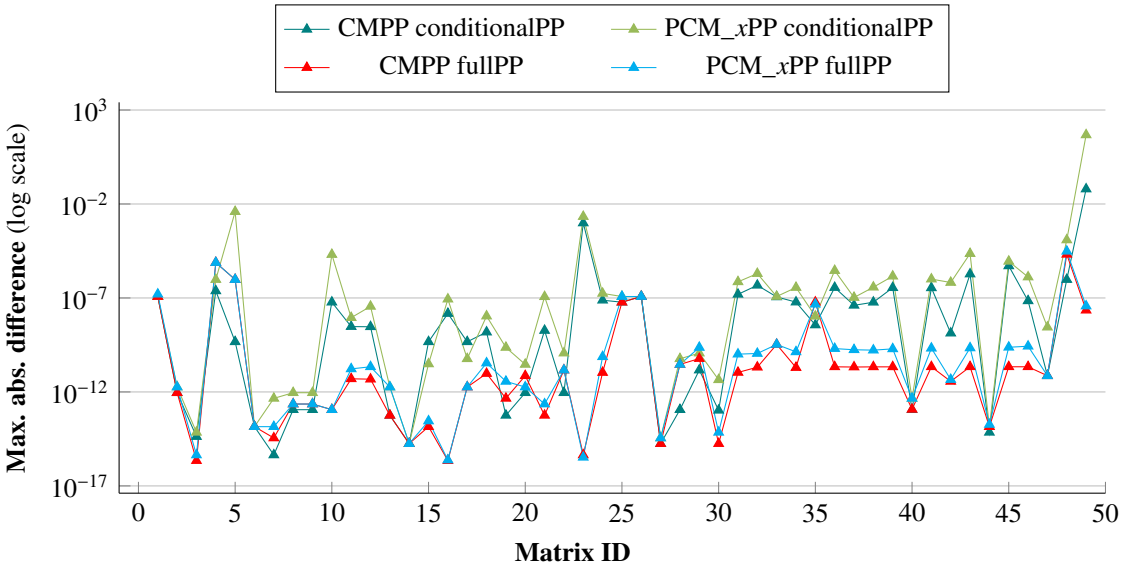


Figure 3.12: Accuracy of results achieved by CMPP and PCM_*x*PP with different types of partial pivoting on the entire set of 50 matrices using double precision. The name of each decomposer follows the naming pattern *decomposer <type_of_pivoting>*. The vertical axis is log-scaled for improved visibility. Note that the accuracy values for the first matrix are excluded from the plot as the results produced by the decomposers matched the expected results.

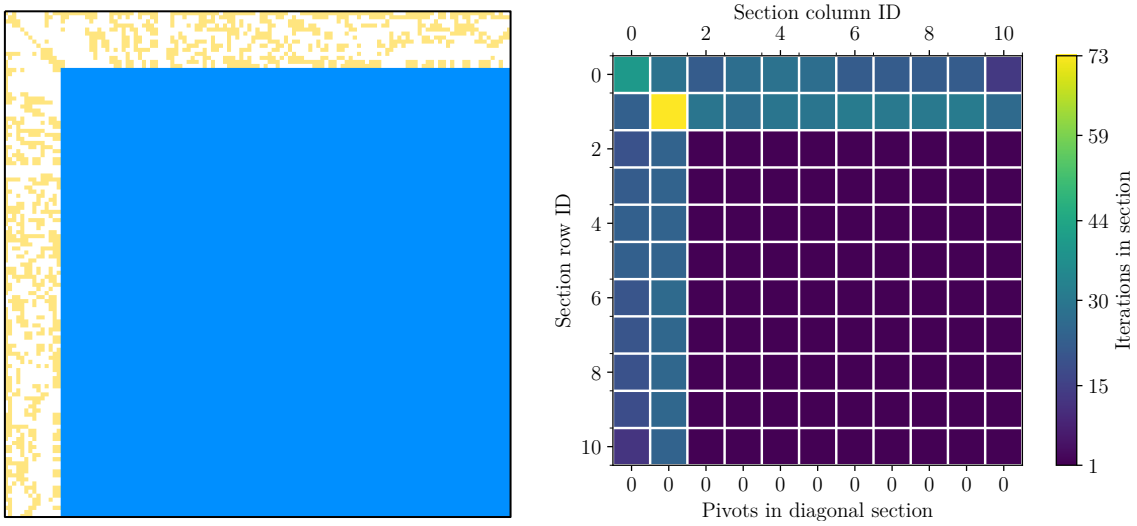
As shown in Figure 3.12, CMPP and PCM_*x*PP produced less accurate results with conditional partial pivoting than with full partial pivoting. This decrease in accuracy would be concerning if CMPP and PCM_*x*PP were intended to be fully-functional decomposers. However, as previously mentioned, both decomposers primarily served as a proof-of-concept for conditional partial pivoting and as a baseline comparison for ICM_*x*PP.

In the introduction of *Processing by Sections* in Section 2.2.2, the upper bound for conditional partial pivoting was introduced. It was explained that the upper bound is necessary only for ICM_*x*PP, as without it, the decomposer would decompose certain matrices incorrectly. In this context, the version of ICM_*x*PP without the upper bound for conditional partial pivoting will be referred to as ICM_*x*PP_NaN. Specifically, ICM_*x*PP_NaN would produce results that contained `-NaN` values in seven of the 50 matrices. The only common characteristic of the matrices was that they were all sparse. To illustrate this issue, Figure 3.13 depicts the nonzero element pattern of the `LU` matrix produced by ICM_32PP_NaN through the decomposition of the *msc10848* matrix. The figure also displays the iterative and pivoting metrics of the decomposer that were collected during the decomposition of the aforementioned matrix.

As depicted in Figure 3.13a, the resulting matrix consisted mostly of `-NaN` values. When considering the metrics presented in Figure 3.13b, it can be seen that ICM_32PP_NaN erroneously classified the sections containing `-NaN` values as processed.

This issue was discovered while plotting the nonzero element patterns of the `LU` matrices produced by ICM_32PP_NaN. The unit tests and benchmark results did not reveal any indications of invalid values. The maximum absolute difference was calculated as follows: `TNL::max( TNL::abs( A - A_res ) )`, where `A_res` represents the result of `L*U`. The value of the maximum absolute difference did not indicate the presence of `-NaN` values, as the `TNL::abs()` function returned `-NaN`.

Furthermore, as depicted in Figure 3.13b, the dark blue sections were processed in a single iteration.



(a) Nonzero element pattern of the `LU` matrix produced by ICM_32PP_NaN through the decomposition of the *msc10848* matrix. The blue square indicates the presence of `NaN` values (`-NaN` in absolute value).

(b) Iterative and pivoting metrics of ICM_32PP_NaN collected during the decomposition of the *msc10848* matrix. Each square represents a 1024-by-1024 section of the matrix. The color in each square represents the number of iterations performed on that section. The left and top axes show the section ID in each dimension, while the bottom axis shows the number of pivots performed in a diagonal section.

Figure 3.13: Nonzero element pattern of the `LU` matrix produced by ICM_32PP_NaN through the decomposition of the *msc10848* matrix. The corresponding iterative and pivoting metrics of ICM_32PP_NaN are included.

This implies that during the processing check, as shown on Page 58 in Listing 2.9 on Line 93, the `-NaN` values were incorrectly evaluated as processed. Specifically, the condition `( abs( val - NaN ) > process_tol )` was evaluated as `false` due to the `abs()` function also returning `-NaN`.

Nevertheless, how ICM_*x*PP_NaN computed the `-NaN` values remains unclear. However, it was observed that the introduction of the upper bound for conditional partial pivoting resolved the issue in all affected matrices. Although the inclusion of the upper bound had a negative impact on the performance of ICM_*x*PP on those particular matrices, it did not affect the performance of the decomposer on the remaining matrices. It is hypothesized that the use of the upper bound for conditional partial pivoting serves as a temporary checkpoint in the computation, effectively eliminating any potential issues that could result in the computation of `-NaN` values. However, it is also possible that the introduction of the upper bound inadvertently masks another underlying issue in the implementation.

While the issue seems to have been resolved by introducing the upper bound for conditional partial pivoting, further investigation should be conducted to gain a better understanding of the underlying causes. Additionally, to prevent similar issues from going unnoticed in the future, the result verification process for the Decomposition benchmark for decomposers ought to be revised. The discovery of this issue revealed that the maximum absolute difference is not a reliable measure of accuracy, as it does not consider the relative accuracy of results. For example, a maximum absolute difference of 1e−4 may appear reasonable, however, if the values in the input matrix are small, such as 1e−8, then the resulting decomposition is highly inaccurate.

Single Precision The statistical indices of the accuracy of the results obtained using single precision are presented in Table 3.9.

Decomposer	Mean	Std. dev.	Q1	Q2	Q3	Max.
CMPP	6.35e+3	3.73e+4	6.10e−5	1.717	64.000	2.62e+5
CuSolverDnXgetrfPP	7.49e+2	4.66e+3	9.01e−5	0.016	0.227	3.28e+4
ICM_*x*PP	6.37e+4	3.29e+5	2.08e−3	15.671	299.311	2.10e+6
PCM_*x*PP	4.96e+5	2.67e+6	0.002	27.828	565.789	1.84e+7

Table 3.9: Statistical indices of the accuracy of the results produced by the decomposers in this benchmark using single precision. Columns Q1, Q2, and Q3 represent the first, second, and third quartiles, respectively. The indices were computed using LibreOffice Calc, a spreadsheet software.

Overall, the decomposers were less accurate when using single precision. From Table 3.9, it can be seen that the mean maximum absolute difference was higher for the majority of decomposers compared to when double precision was used. However, the values of the first and third quartiles suggest that the mean values are heavily influenced by 12 to 13 of the 50 matrices.

When using single precision, the most accurate results were produced by the CuSolverDnXgetrfPP decomposer. Conversely, the remaining decomposers produced relatively highly inaccurate results. Excluding CuSolverDnXgetrfPP, the most accurate decomposer was CMPP. However, similar to double precision, the accuracy of the results produced by CMPP suffered due to conditional partial pivoting. The accuracy of the decomposer was higher when full partial pivoting was used. The comparison of the accuracy of results between CMPP and PCM_*x*PP, considering both conditional partial pivoting and full partial pivoting, is presented in Figure 3.14.

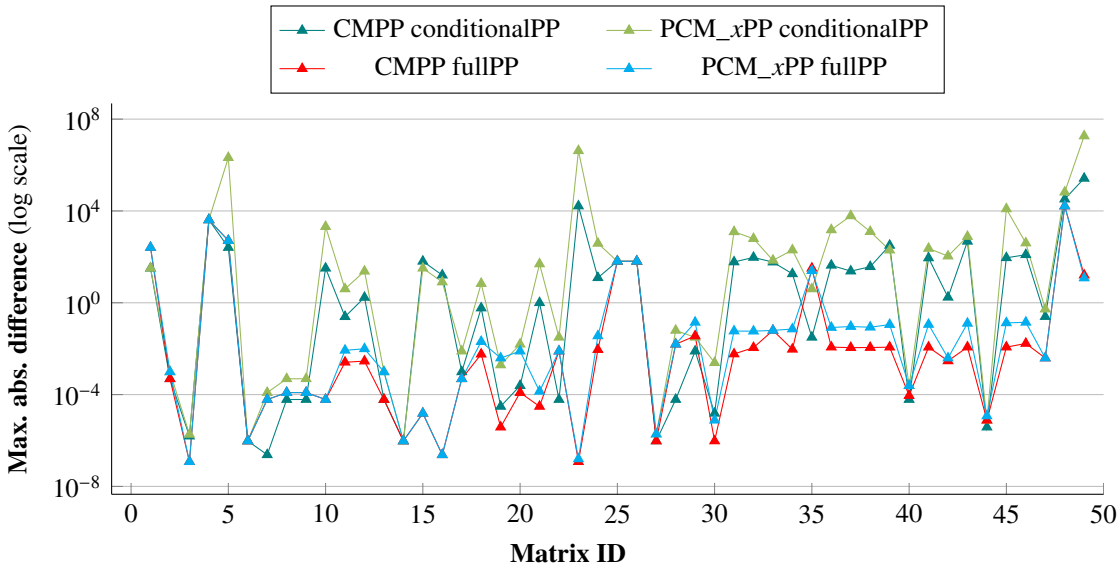


Figure 3.14: Accuracy of results achieved by CMPP and PCM_*x*PP with different types of partial pivoting on the entire set of 50 matrices using single precision. The naming pattern for each decomposer is *decomposer <type_of_pivoting>*. The vertical axis is log-scaled for better visibility. Note that the accuracy values for the first matrix are excluded from the plot as the results produced by the decomposers matched the expected results.

Figure 3.14 highlights the instability of conditional partial pivoting. Specifically, it demonstrates that the difference in accuracy between CMPP conditionalPP and PCM_xPP conditionalPP is much greater with single precision compared to double precision. These results indicate that conditional

partial pivoting, when combined with lower precision, leads to an unstable approach that produces inaccurate results.

To summarize the accuracy of results on all matrices, the decomposer that consistently produced the most accurate results regardless of the precision used was CuSolverDnXgetrfPP. Excluding CuSolverDnXgetrfPP, CMPP is recommended for both double and single precision in terms of accuracy. However, if high accuracy is required, then CMPP with full partial pivoting is more suitable compared to CMPP with conditional partial pivoting.

Summary of Decomposers Benchmark The fastest and most accurate decomposer on the set of 50 matrices was CuSolverDnXgetrfPP. Among the decomposers implemented by the author of this thesis, PCM_8PP is a suitable choice if execution speed is the primary requirement, as it consistently performs well. However, for sparse matrices with most of their elements on the main diagonal, ICM_32PP can be a suitable alternative. Furthermore, for specific problems, such as the "Poisson equation on a cube" problem implemented in BDDCML, ICM_32PP may outperform PCM_8PP. On the other hand, if highly accurate results are required, CMPP is recommended, even at the cost of execution speed.

3.1.4 Solvers Benchmark

This section presents benchmark results obtained by solving systems of linear equations that originated from the set of 50 matrices. The benchmark for solvers consists of each matrix being used as a coefficient matrix $\mathbf{A}$ in $\mathbf{AX} = \mathbf{B}$. Specifically, the benchmark comprises the following steps:

1. The matrix is decomposed either by CMPP or GEM depending on which of the two matrices, `L` or `U`, is expected by the solver to contain a unit diagonal. If the unit diagonal is expected in `U`, then CMPP is used, whereas if it is expected in `L`, then GEM is used.
2. To mitigate possible inaccuracies caused by the decomposer, matrices `L` and `U` are multiplied to create `A_res` which is later used instead of `A` for the verification of results.
3. The matrix of 10 right-hand sides, `B`, is filled with ones.
4. The solver receives the decomposed matrices, `L` and `U`, the pivoting vector `piv`, the matrix of unknowns, `X`, and the matrix of right-hand sides, `B`.
5. The system is solved 10 times by the solver, and the collected metrics are subsequently averaged and logged.

The solvers compared in this benchmark are listed in Table 3.10.

Note that the processing tolerance of IS_*x*PP was set to zero, as higher accuracy of the produced results is required. As mentioned in Section 2.2.2, the implementation of solvers was not part of the diploma thesis assignment. Furthermore, neither SSPP nor IS_*x*PP were prioritized in terms of optimization. Therefore, the benchmark results of the solvers will only be presented briefly for the entire set of 50 matrices.

First, the execution times of the solvers listed in Table 3.10 on the set of 50 matrices are presented for both double and single precision. Then, the accuracy of the results is discussed.

Solver		Variants	Performer
Name	Abbreviation		
Sequential Solver with Partial Piv.	SSPP	-	CPU
CuBLAStrsm with Partial Piv.	CuBLAStrsmPP	-	GPU
CuSolverDnXgetrs with Partial Piv.	CuSolverDnXgetrsPP	-	GPU
Iterative Solver with Partial Piv.	IS_*x*PP	8, 16, 32, 64, 128	GPU

Table 3.10: Solvers compared in the benchmark: SSPP (described in *Sequential Solver with Partial Pivoting (SSPP)* in Section 2.2.2), CuBLAStrsmPP (mentioned in Section 2.1.2), CuSolverDnXgetrsPP (mentioned in Section 2.1.2), and IS_*x*PP (described in *Iterative Solver with Partial Pivoting (IS_xPP)* in Section 2.2.2). The *Variants* column indicates the different configurations of each solver. The IS_*x*PP solver was benchmarked as five separate solvers depending on the CUDA thread block configuration. The *x* in IS_*x*PP represents the number of threads per block in the 1st dimension. The *Performer* column specifies the primary device used by each solver. The CuBLAStrsmPP solver assumed that the unit diagonal was present in matrix `U`.

Comparison of Execution Times on All Matrices

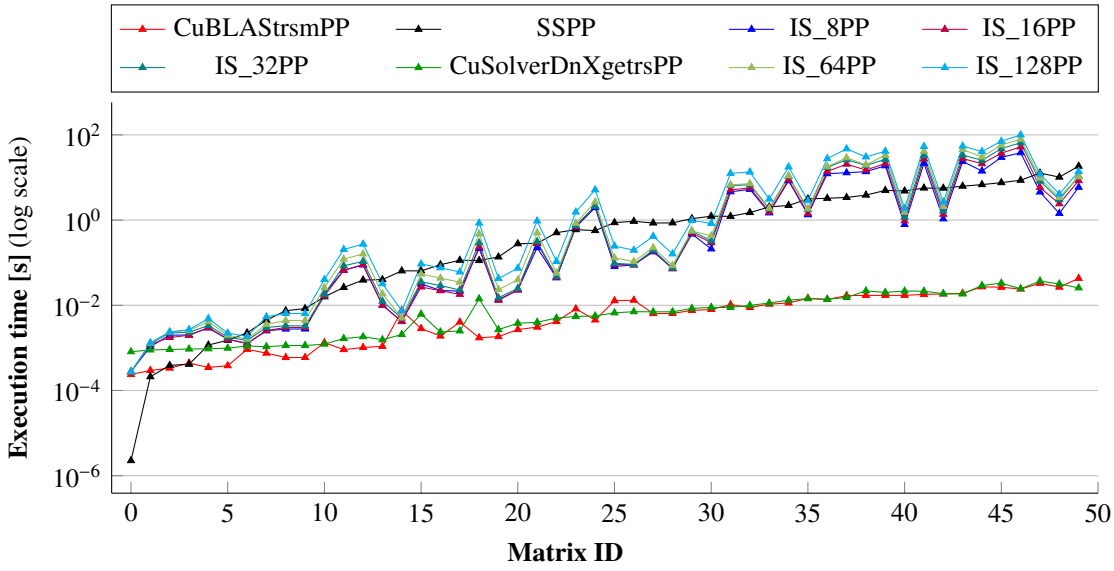
The comparison of execution times of the solvers listed in Table 3.10 on the set of 50 matrices using both double and single precision is presented in Figure 3.15.

As depicted in Figure 3.15, CuBLAStrsmPP and CuSolverDnXgetrsPP were the fastest solvers overall, regardless of the precision used. The results indicate that, similar to ICM_*x*PP, the performance of the iterative approach of IS_*x*PP depends on the characteristics of the matrix. For example, when double precision was used, IS_*x*PP outperformed SSPP on 26 out of the 50 matrices. These 26 matrices were made up of 19 sparse matrices from *The university of Florida sparse matrix collection* [53] and seven dense matrices from the "Poisson equation on cube" problem implemented in BDDCML. When considering matrices with dimensions up to approximately 3600-by-3600 (matrices 0 to 29), it can be seen that the execution times of IS_*x*PP are closer to those of the established CUDA solvers compared to matrices with larger dimensions. This suggests that the performance of IS_*x*PP and SSPP does not scale well.

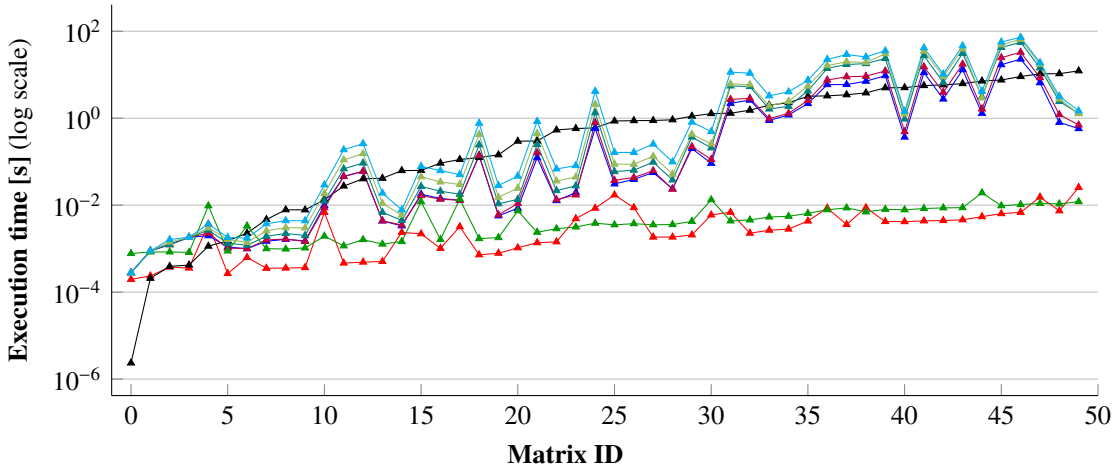
Additionally, when examining the variants of IS_*x*PP, it is evident that IS_128PP is consistently surpassed by variants with a smaller number of threads per block, for both double and single precision. To confirm this observation, Figure 3.16 presents a comparison of execution times for the IS_*x*PP variants on the set of matrices when double precision was used.

As depicted in Figure 3.16, IS_8PP consistently demonstrated better performance compared to IS_16PP, which in turn consistently outperformed IS_32PP, and so on. These findings suggest that IS_*x*PP is also affected by the resource idleness issue described on Page 78 for PCM_*x*PP. In IS_*x*PP, the thread blocks comprise *x*-by-8 threads, as shown in Listing E.1 on Line 52. Each row of threads within the eight rows of threads in the thread block is responsible for computing values for one right-hand side. The number of rows in the thread block has remained constant since the initial implementation of the solver. As solvers were not a part of the diploma thesis assignment, the implementation of thread blocks that would dynamically adapt to the number of right-hand sides was not prioritized. Each thread in the 1st dimension is responsible for computing one element of a right-hand side. For example, thread `(x = 10, y = 5)` is responsible for computing the 10th element of the 5th right-hand side.

To aid the explanation of the resource idleness that arises in IS_*x*PP as a result of its thread block structure, the visualizations of forward and backward substitution are shown in Figure 3.17.



(a) Double precision



(b) Single precision

Figure 3.15: Execution times (in seconds) of solvers listed in Table 3.10 on the set of 50 matrices using double and single precision. The vertical axis is log-scaled for improved visibility.

For simplicity, hereafter, the substitutions are assumed to work with only one right-hand side, unless stated otherwise. In Figures 3.17a and 3.17b, each row represents the computation of a value by its assigned thread. Since a warp comprises 32 threads, each warp is responsible for computing 32 adjacent elements. Starting with the first thread of the warp, each subsequent thread performs an additional calculation. For example, thread `0` computes the value with one calculation, while thread `1` computes the value with two calculations, and so on. Therefore, the time it takes for each warp to complete its work depends on the thread with the most calculations to perform, i.e., the last thread.

The number of threads per block for IS_8PP, IS_16PP, IS_32PP, IS_64PP, and IS_128PP is 64, 128, 256, 512, and 1024, respectively. The maximum number of thread blocks per SM for each variant is 32, 16, 8, 4, and 2, respectively. For example, IS_128PP can only allocate two thread blocks per SM. Since each thread block covers the computation of 128 elements, the thread computing the last element has to perform an additional 128 calculations compared to the first thread of the block. Therefore, while the threads of the last warp are still computing their elements, the warps preceding the last warp are idle as they will have completed their work earlier. In other words, the last warp

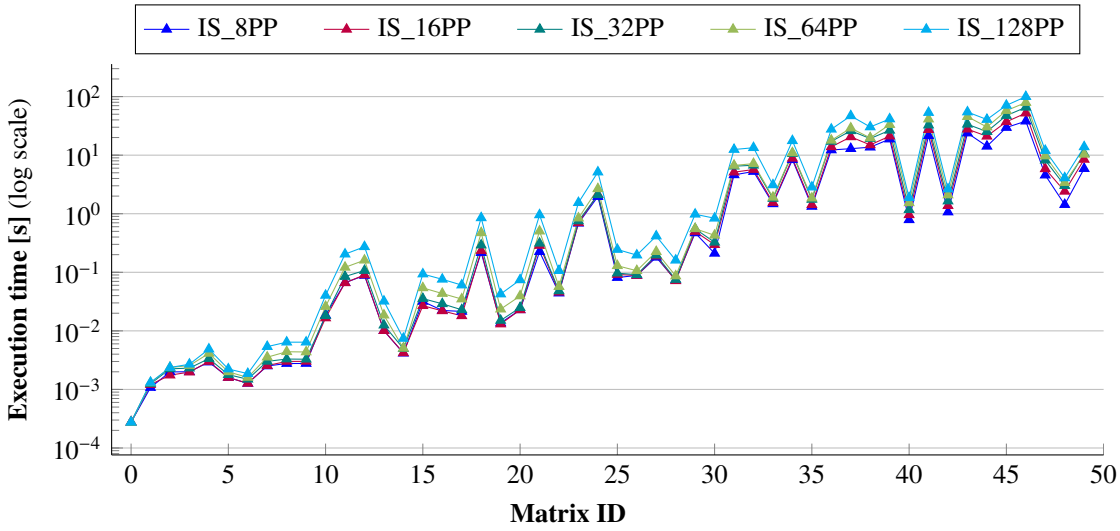
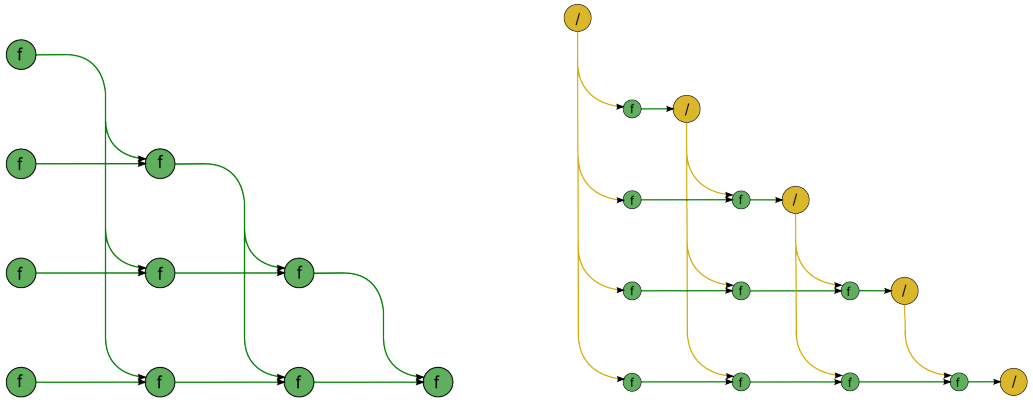


Figure 3.16: Execution times (in seconds) of the variants of IS_*x*PP on the set of 50 matrices using double precision. The vertical axis is log-scaled for improved visibility.



(a) Forward substitution

(b) Backward substitution

Figure 3.17: Visualizations of the forward and backward substitution algorithms. The visualizations assume a single right-hand side. Each row in both figures represents the computation of one value in the output vector of the substitution. The arrows indicate the use of previously-computed values in the computation of succeeding values. Taken from *Backward substitution* [60] and *Forward substitution* [61].

of each thread block holds the thread block in an SM, preventing the scheduling of another thread block.

As with PCM_*x*PP, the variants of IS_*x*PP with fewer threads per block were able to utilize SMs more efficiently, as more of their smaller blocks could be allocated per SM. It can be argued that the idleness of resources is further exacerbated by the fixed size of the 2nd dimension of the thread block, which is set to eight. If the number of threads per block were to adapt to the number of right-hand sides, the number of threads in each block would decrease, enabling the allocation of more blocks per SM.

To put the results into perspective, Table 3.11 displays the total time taken by each solver to solve systems derived from the entire set of 50 matrices.

Solver	Total execution time [s]	
	Double precision	Single Precision
SSPP	120.963	114.645
CuBLAStrsmPP	0.479	0.209
CuSolverDnXgetrsPP	0.505	0.274
IS_128PP	561.879	414.728
IS_16PP	282.535	156.937
IS_32PP	351.020	279.547
IS_64PP	415.793	334.692
IS_8PP	224.219	115.321

Table 3.11: Total execution time (in seconds) taken by each solver to solve systems derived from the set of 50 matrices using double and single precision. The execution times are rounded to three decimal places.

As demonstrated in Figure 3.15 and confirmed in Table 3.11, the fastest solver for the set of 50 matrices was CuBLAStrsmPP. Excluding the solvers from CUDA libraries, SSPP was the fastest solver, even though it outperformed IS_8PP on only 24 out of 50 matrices.

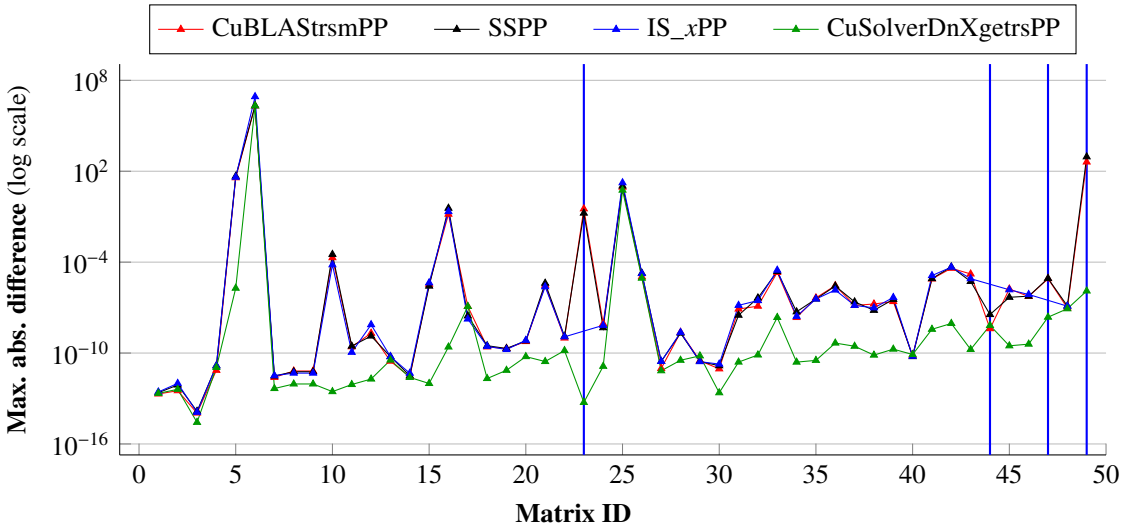
Accuracy of Results on All Matrices

The accuracy of the values of unknowns produced by solvers is a crucial performance indicator. The maximum absolute difference between the expected results and the actual results for the set of 50 matrices using both double and single precision is presented in Figure 3.18. For context, the maximum absolute difference for solvers was defined in Equation 2.8 as $\max |\mathbf{LUPX} - \mathbf{B}|$, where **L** (lower-triangular matrix), **U** (unit upper-triangular matrix), and **P** (permutation matrix) are the results of the decomposition operation performed on the input matrix; **X** represents the matrix of unknowns whose values are computed by a solver, and **B** denotes the matrix of right-hand sides (with all values equal to 1). Note that the permutation matrix was represented by a vector in the implementations. This section mentions the IS_*x*PP solver instead of its variants, as they did not differ in terms of accuracy, i.e., there was no need to specifically address each variant.

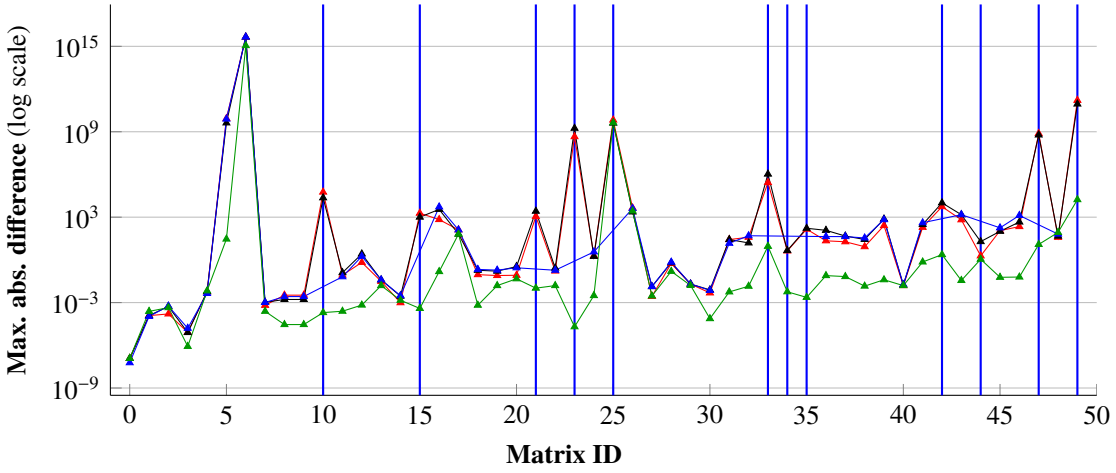
As mentioned in the caption of Figure 3.18, IS_*x*PP occasionally produced invalid results. Specifically, when using double precision, for four matrices, the solver produced results that resulted in the maximum absolute difference being equal to 1.797 693e+308, which is the highest possible value that a variable of type `double` can store. In the case of single precision, the solver produced invalid results for twelve matrices, with the maximum absolute difference equal to -3.402 823e+38 which is close to the lowest possible value that a variable of type `float` can store. Among the twelve matrices, eleven were sparse, and one was dense. Additionally, it is worth noting that the CPU version of IS_*x*PP also produced the same invalid results, indicating that the issue lies with the algorithm itself rather than the GPU implementation. The reason behind IS_*x*PP computing invalid values in the mentioned cases is currently unclear. One possible hypothesis is that the computation of certain values is being skipped, leading to a cascade of erroneous calculations.

For completeness, the statistical indices of the accuracy of the results produced by the solvers are presented in Table 3.12 for double precision and in Table 3.13 for single precision. Note that both tables contain two instances of IS_*x*PP: IS_*x*PP and IS_*x*PP_inv. The former does not consider the invalid values in its indices, while the latter does.

As observed in Tables 3.12 and 3.13, the accuracy of the results decreased significantly from double



(a) Double precision



(b) Single precision

Figure 3.18: Accuracy of results achieved by the solvers listed in Table 3.10 (excluding the different variants) on the entire set of 50 matrices using both double and single precision. The matrix ID represents the ID of the matrices after they have been sorted according to their dimensions from smallest to largest. The vertical axis is log-scaled for improved visibility. The blue vertical lines indicate matrices for which IS_*x*PP produced invalid results. The accuracy values for the invalid results were not plotted to avoid distorting the vertical axis and allow for comparison of the remaining accuracy values. Additionally, it is important to mention that the vertical axes of both graphs do not cover the same range.

to single precision. Specifically, focusing on double precision and excluding IS_*x*PP_inv, it can be seen from the 3rd quartiles and the maximum values that the mean values were heavily offset by 12 to 13 matrices. For example, except for IS_*x*PP_inv, the maximum inaccuracy among all solvers was consistently observed with the 156-by-156 matrix *west0156*. This can be attributed to the matrix's high condition number of approximately 6.09e+19, which indicates its sensitivity to numerical errors and makes it more challenging for the solvers to achieve accurate results.

Both Table 3.12 and Table 3.13 show that the accuracy of IS_*x*PP, i.e., excluding the invalid results, is not poor relative to the other solvers. However, as the issue with its implementation causes it to produce invalid values, its actual performance in terms of accuracy renders it unusable. This

Solver	Mean	Std. dev.	Q1	Q2	Q3	Max.
SSPP	4.20e+4	2.97e+5	5.82e−11	4.38e−8	4.99e−6	2.10e+6
CBLAStrsmPP	4.20e+4	2.97e+5	4.08e−11	8.75e−8	6.70e−6	2.10e+6
CSDnXgetsfPP	4.19e+4	2.97e+5	1.96e−12	4.63e−11	4.39e−10	2.10e+6
IS_*x*PP	1.82e+5	1.24e+6	3.55e−11	2.20e−8	2.16e−6	8.39e+6
IS_*x*PP_inv	1.43e+307	4.93e+307	1.56e−11	7.12e−9	1.49e−6	1.80e+308

Table 3.12: Statistical indices of the accuracy of the results produced by the solvers in this benchmark using *double* precision. Columns Q1, Q2, and Q3 represent the first, second, and third quartiles, respectively. To avoid formatting issues, the names of the CuSolverDnXgetrsPP and CuBLAStrsmPP solvers are abbreviated. The indices were computed using LibreOffice Calc, a spreadsheet software.

Solver	Mean	Std. dev.	Q1	Q2	Q3	Max.
SSPP	9.01e+13	6.37e+14	0.025	23.018	1.37e+3	4.50e+15
CuBLAStrsmPP	9.01e+13	6.37e+14	0.023	13.568	672.659	4.50e+15
CuSolverDnXgetrsPP	2.25e+13	1.59e+14	6.57e−4	0.016	0.155	1.13e+15
IS_*x*PP	1.19e+14	7.31e+14	0.009	0.484	112.875	4.50e+15
IS_*x*PP_inv	8.17e+37	1.47e+38	3.98e−6	0.030	43.327	8.17e+37

Table 3.13: Statistical indices of the accuracy of the results produced by the solvers in this benchmark using *single* precision.

conclusion is supported by the statistical indices of IS_*x*PP_inv.

To summarize, the most accurate solver for both double and single precision on the set of 50 matrices was CuSolverDnXgetrsPP, followed by CuBLAStrsmPP. The current implementation of IS_*x*PP cannot be recommended for use as it contains an unknown issue.

Taking into account both execution time and the accuracy of the results, the best-performing solver is CuSolverDnXgetrsPP. It exhibits only a marginal increase in execution time compared to CuBLAStrsmPP while producing more accurate results.

3.1.5 Summary of the Results of the Decomposition Benchmarks

Overall, in terms of both execution speed and accuracy, the decomposers and solvers provided by the CUDA libraries outperformed the others. Specifically, the CuSolverDnXgetrfPP decomposer demonstrated the fastest execution speed while producing the most accurate results. Although the CuSolverDnXgetrsPP solver was marginally slower than CuBLAStrsmPP, it produced more accurate results. Among the decomposers implemented by the author of this thesis, PCM_*x*PP emerged as the fastest and most accurate overall. However, for specific types of matrices, the ICM_*x*PP decomposer may be more suitable compared to PCM_*x*PP. Additionally, the processing tolerance of ICM_*x*PP may prove useful if the decomposer is used as a preconditioner that is not expected to produce highly accurate results. While the benchmark results of the IS_*x*PP solver confirmed that the iterative approach can save time for certain matrix types, its implementation contains an unidentified issue and thus it was disqualified.

3.2 BDDCML Benchmark

This section presents the benchmark results of the "Poisson equation on a cube" problem implemented within BDDCML, as introduced in Section 2.3. The benchmark was conducted only using double precision, as single precision was not supported by the problem at the time. Additionally, the benchmark was run only for a selection of decomposers, i.e., the performance of solvers was not compared. This decision was made based on two reasons: firstly, the solvers implemented by the author of this thesis for the GPU were deemed unusable, and secondly, comparing the performance of solvers from CUDA libraries with the naive CPU implementation of SSPP would not bring any meaningful insights.
For each decomposer, the benchmark involved the following steps:

1. Perform a warm-up by running the `poisson_on_cube` executable with undemanding parameters.
2. For each value in the set $\{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$, run the `poisson_on_cube` executable 10 times, providing the value as the first argument (`poisson_on_cube <value> <arg2> <arg3>`). Note that the decomposer is called multiple times within each run.
3. Collect and save the logs generated by each run, as they contain the metrics relevant to this benchmark.

The logs are then parsed to obtain the metrics which are subsequently averaged using the Python script mentioned in Section 2.3. While the `poisson_on_cube` executable accepts three parameters, only the value of the first parameter varied throughout the benchmark. The parameters of the executable are as follows:

1. `num_el_per_sub_edge` - The number of elements per sub-edge of the cube. In terms of BDDCML, it is denoted by H/h, where H is the size of an edge and h represents the size of the edge of an element [62]. H/h is also referred to as the *subdomain size* [62].
2. `num_sub_per_cube_edge` - The number of sub-edges per cube edge. For this benchmark, the value was set to 4.
3. `nlevels` - The number of levels in the BDDC method. For this benchmark, the value was set to 2.

In brief, the "Poisson equation on a cube" problem in BDDCML is divided into three stages that are timed separately:

1. *loading* - This stage involves loading the data for the problem.
2. *pc_setup* - The *Preconditioner Setup* stage includes the utilization of a decomposer as a preconditioner along with a solver.
3. *Krylov method* - This stage involves employing matrix-vector multiplication alongside a solver to iteratively improve the solution. The number of iterations required for the Krylov method to reach the desired relative residual serves as an indicator of accuracy for the decomposer utilized in the *pc_setup* stage, as the matrices used in this stage are provided by the preconditioner.

In this section, only the last two stages will be mentioned as they utilize the procedures (decomposers and solvers) provided by the Decomposition project. Similar to the Decomposition project benchmarks, only the variants of the procedures with partial pivoting were employed.

Firstly, the specifications of the platform on which the benchmark was executed are provided, as they slightly differ from those of the system used for the Decomposition project benchmarks. Then, the types of matrices encountered by the decomposers in the benchmarks are described. Finally, the performance of decomposers is analyzed.

3.2.1 Benchmark Platform Specifications

Similarly to the benchmarks of the Decomposition project, the BDDCML benchmark was also run on the RCI cluster. However, due to the higher GPU memory requirements of the problem, particularly with larger values of H/h, a different compute node was used. Although the node also contained an AMD CPU and an Nvidia GPU, the hardware configuration differed. Furthermore, adjustments had to be made to the configuration since BDDCML relies on multiple other libraries. Refer to Table 3.14 for the hardware and software configuration.

Hardware			
Component	**Specification**		
CPU	AMD EPYC 7763 @ 2.45GHz (32 cores, 64 threads)		
Memory	1000GB		
GPU	8x Nvidia Tesla A100 40GB		
Software			
Library/Tool	**Version**	**Library/Tool**	**Version**
CMake[4]	3.24.3	MUMPS[5]	5.6.0
GCC[6]	12.2.0	OpenBLAS[7]	0.3.21
CUDA[8]	12.0.0	Open MPI[9]	4.1.4
MAGMA[10]	2.7.1	ParMETIS[11]	4.0.3
METIS[12]	5.1.0	ScaLAPACK[13]	2.2.0

Table 3.14: Hardware and software configuration of the RCI cluster node on which the BDDCML benchmark was run. The information is sourced from *RCI Cluster Wiki* [52] and from the computation job's configuration. The hardware configuration is specified in a *batch* script, which includes SLURM-specific job settings and Bash code for the compute node to execute. The dependencies were either specified in the batch file as modules or provided as directories. The benchmark scripts are available on request or as an attachment to this thesis.

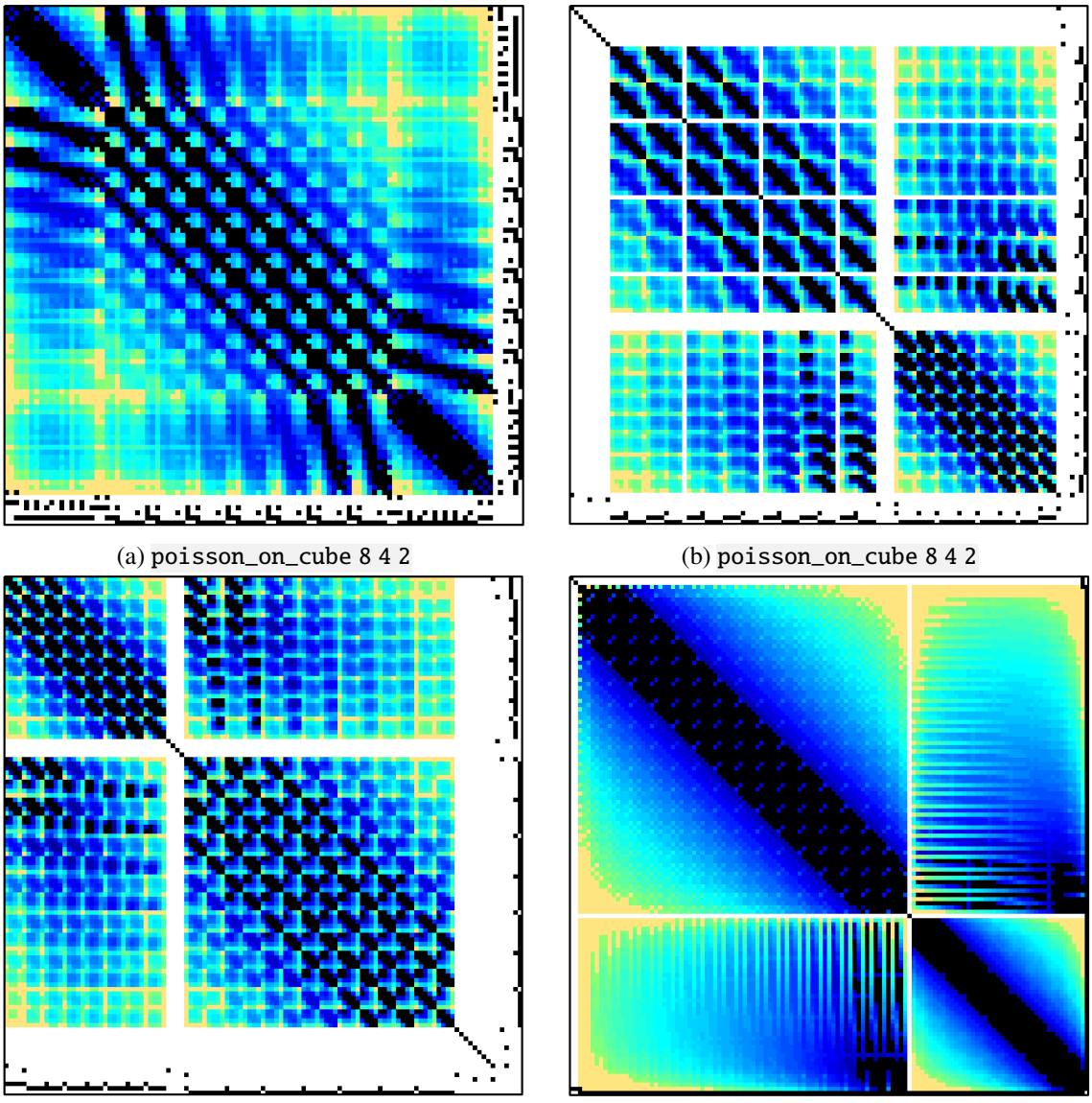
It is important to mention that the executable was run using `srun` with `pmix`[14], which is the standard SLURM command to start an MPI program. In particular, the number of MPI ranks launched to execute the `poisson_on_cube` program was 32. Furthermore, the ranks were distributed equally across the eight GPUs available using the modulo operation: `my_device = my_rank % 8`.

[4]CMake website URL: `https://cmake.org`
[5]MUMPS website URL: `https://mumps-solver.org/index.php`
[6]GCC website URL: `https://gcc.gnu.org`
[7]OpenBLAS website URL: `https://www.openblas.net`
[8]CUDA website URL: `https://developer.nvidia.com/cuda-toolkit`
[9]Open MPI website URL: `https://www.open-mpi.org`
[10]MAGMA website URL: `https://icl.utk.edu/magma`
[11]ParMETIS website URL: `https://github.com/KarypisLab/ParMETIS`
[12]METIS GitHub repository URL: `https://github.com/KarypisLab/METIS`
[13]ScaLAPACK website URL: `https://netlib.org/scalapack`
[14]PMIX website URL: `https://pmix.github.io`

3.2.2 Matrices in the Benchmark

As mentioned before, the set of values used for H/h was limited to the interval $[5, 50]$. The values in the set determine the size of the matrices passed to the decomposer. The largest value in the interval, 50, was chosen as the ICM_*x*PP decomposer would run out of GPU memory for higher values of H/h since the matrices were stored in column-major order on the GPU.

Given the values of H/h, the dimensions of the matrices that decomposers factorized during the *pc_setup* stage ranged from 104-by-104 to 15,028-by-15,028. As mentioned earlier, the decomposer can be called multiple times during the execution of `poisson_on_cube`. The dimensions of the matrices provided to the decomposer can vary slightly within the same run. For example, for `poisson_on_cube 5 4 2`, the dimensions of the matrices ranged from 104-by-104 to 178-by-178. The nonzero element pattern of the matrices was similar even with different values of H/h. While the matrices were mostly dense, they occasionally contained patches of zeros, as shown in Figure 3.19.



(a) `poisson_on_cube 8 4 2`

(b) `poisson_on_cube 8 4 2`

(c) `poisson_on_cube 8 4 2`

(d) `poisson_on_cube 32 4 2`

Figure 3.19: Visualization of the nonzero element patterns in matrices encountered in the BDDCML benchmark. The captions indicate the configuration in which each matrix appeared.

In Figure 3.19, matrices with different element patterns can be observed in the same configuration: `poisson_on_cube 8 4 2`. Furthermore, while the nonzero element pattern in Figure 3.19d appeared in `poisson_on_cube 32 4 2`, it seems to be a denser representation of the nonzero element pattern displayed in Figure 3.19b, which appeared in `poisson_on_cube 8 4 2`.

3.2.3 Benchmark Results

This section presents the benchmark results obtained by running the BDDCML benchmark using the decomposers listed in Table 3.15, paired with the CuBLAStrsmPP solver (except for the MAGMA decomposer, which uses its own solver). CuBLAStrsmPP was chosen as it supports the unit diagonal both in **L** and **U**.

Decomposer		Variants	Performer
Name	Abbreviation		
CuSolverDnXgetrf with Partial Piv.	CuSolverDnXgetrfPP	-	GPU
Iterative Crout's Method with Partial Piv.	ICM_*x*PP	8, 16, 32	GPU
magma_dgetrf_gpu	MAGMAgetrf	-	CPU + GPU
Parallel Crout's Method with Partial Piv.	PCM_*x*PP	8, 16, 32	GPU

Table 3.15: Decomposers compared in the benchmark: CuSolverDnXgetrfPP (mentioned in Section 2.1.2), ICM_*x*PP (described in *Iterative Crout's Method with Partial Pivoting (ICM_xPP)* in Section 2.2.2), MAGMAgetrf (description available in the MAGMA documentation[15]), and PCM_*x*PP (described in *Parallel Crout's Method with Partial Pivoting (PCM_xPP)* in Section 2.2.2). The *Variants* column indicates the different configurations of the decomposer. For example, ICM_*x*PP was benchmarked as three separate decomposers depending on the CUDA thread block configuration. In the case of PCM_*x*PP, the thread blocks are one-dimensional, and the number of threads per block is x^2. Unless stated otherwise, the processing tolerance for ICM_*x*PP was set to zero, and the lower bound and upper bound of conditional partial pivoting tolerance were set to 1e−5 and 1e+5, respectively. For PCM_*x*PP, the conditional partial pivoting tolerance was set to 1e−5. Note that the `magmaf_dgetrf_gpu()` function was already used in BDDCML prior to this thesis and it is not available in the Decomposition project. The *Performer* column specifies the primary device used by each decomposer.

First, the speedup comparison in the *Preconditioner Setup* stage between the baseline implementation in BDDCML, MAGMAgetrf, and the other decomposers listed in Table 3.15 is presented. Then, a comparison of the highest-performing decomposers is provided. Finally, the accuracy of the results is mentioned.

Speedup Comparison of Decomposers in the Preconditioner Setup Stage

This section presents the speedup comparison of the decomposers listed in Table 3.15 relative to MAGMAgetrf. For clarity, the times discussed in this section represent the total time taken by the *pc_setup* stage. Specifically, for each benchmark, the `poisson_on_cube` executable was run ten times for each value of H/h. The recorded times from all runs were subsequently averaged and logged. The speedup comparison is shown in Figure 3.20.

As depicted in Figure 3.20, the *pc_setup* stage was completed in the shortest time when either MAGMAgetrf or CuSolverDnXgetrfPP were used. Specifically, with MAGMAgetrf, the execution

[15]MAGMA Documentation for getrf URL: `https://icl.utk.edu/projectsfiles/magma/doxygen/group__magma__getrf.html`

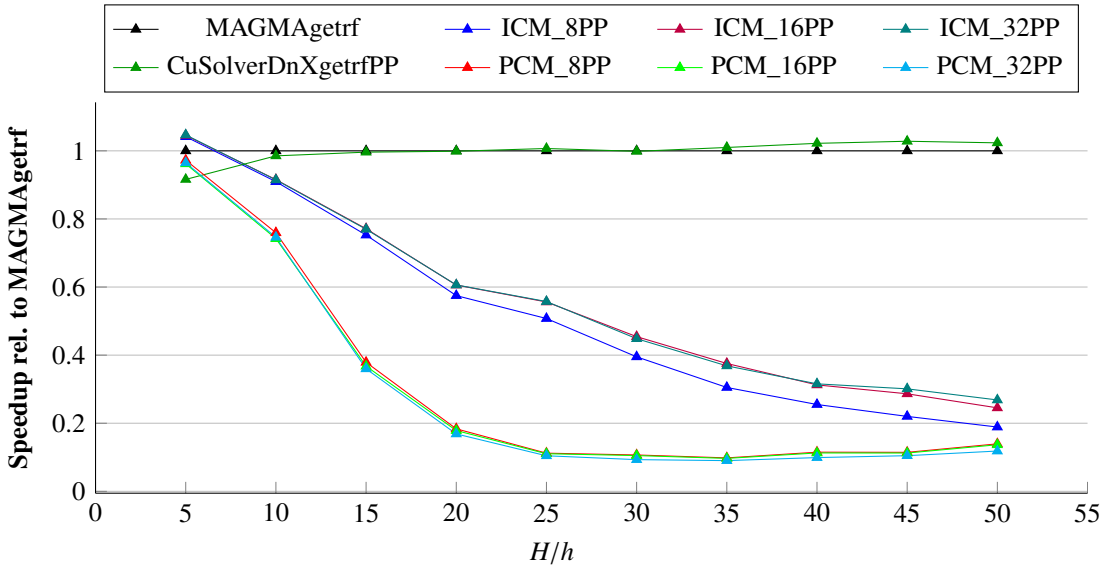


Figure 3.20: Speedup comparison of the *pc_setup* stage for the decomposers listed in Table 3.15 relative to MAGMAgetrf.

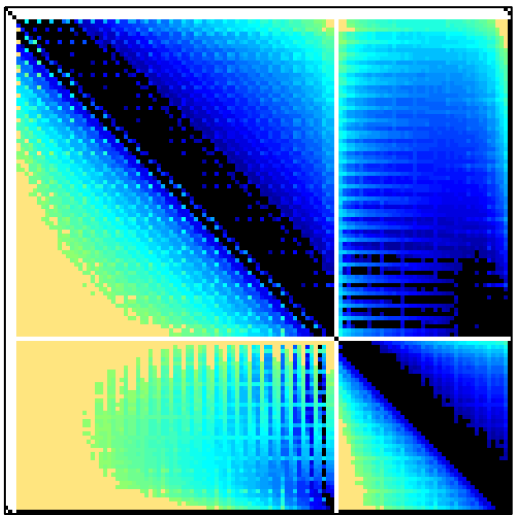
of the stage was faster for smaller values of H/h, whereas for larger values, it was faster with CuSolverDnXgetrfPP. In general, the use of the decomposers implemented by the author of this thesis resulted in the *pc_setup* stage taking longer to complete. Out of the author's decomposers, ICM_*x*PP achieved the best performance.

ICM_*x*PP The best-performing variant of ICM_*x*PP was ICM_32PP. Specifically, ICM_32PP consistently outperformed ICM_16PP, which in turn consistently outperformed ICM_8PP. This performance difference among the variants aligns with the results obtained by the variants of ICM_*x*PP in the Decomposition benchmark for decomposers, as mentioned on Page 77. To further analyze ICM_32PP, Figure 3.21 presents the `LU` matrices ICM_32PP produced, along with its iterative and pivoting metrics for $H/h \in \{25, 50\}$. The matrices presented in Figure 3.21 were chosen at random, as ICM_32PP was called multiple times within each run of the `poisson_on_cube` executable. Incidentally, the nonzero element pattern of both matrices was similar.

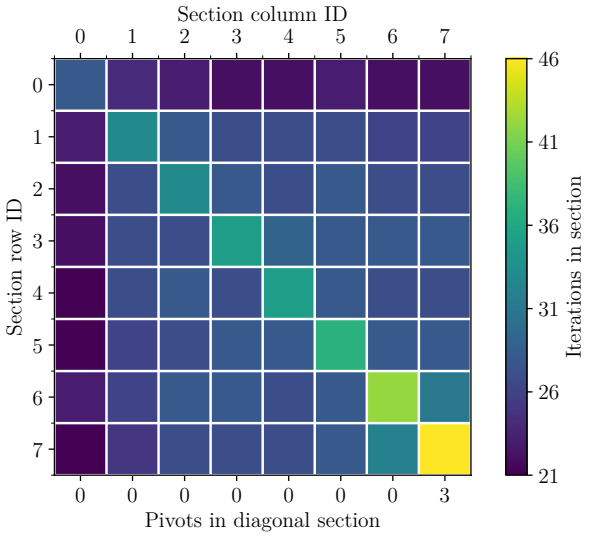
The iterative and pivoting metrics of ICM_32PP presented in Figures 3.21b and 3.21d indicate that the performance can be attributed to three factors:

1. Firstly, the initial estimate of `LU` (the input matrix) was suitable, as indicated by the lower number of iterations for sections with column and row IDs equal to 0. However, the low number of iterations can also be attributed to the patches of zeros, as mentioned in the following point.

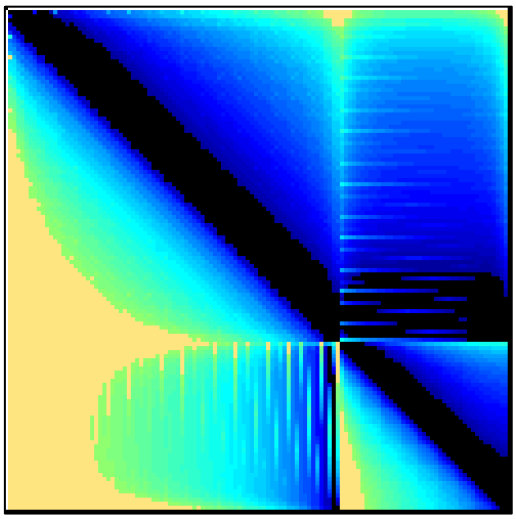
2. Secondly, the matrices contained patches of zeros, which are known to reduce the number of iterations required to process sections. For example, in Figure 3.21d, the sections below and to the right of section (6, 6) required fewer iterations compared to the surrounding sections, as they computed patches of zeros. While the patches are not visible in Figure 3.21c due to the matrix dimensions, they are visible in Figure 3.21a, where the nonzero element pattern is similar.



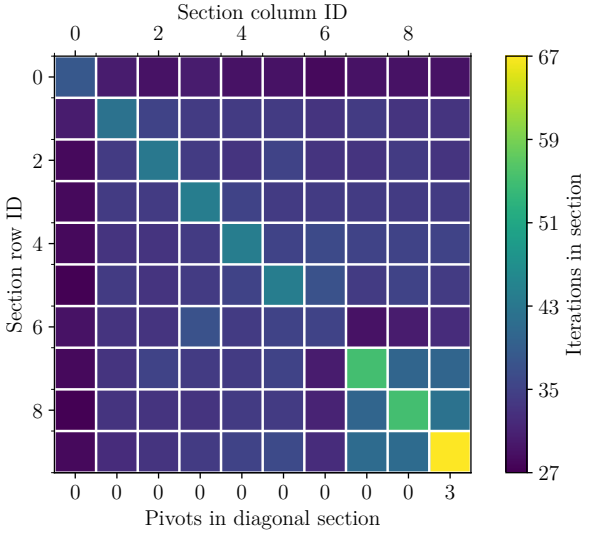
(a) Nonzero element pattern of the LU matrix produced by ICM_32PP for a 1,964-by-1,964 matrix when H/h was set to 25.



(b) Iterative and pivoting metrics of ICM_32PP collected during the decomposition of a 1,964-by-1,964 matrix when H/h was set to 25. Each square represents a 256-by-256 section of the matrix.



(c) Nonzero element pattern of the LU matrix produced by ICM_32PP for a 7,664-by-7,664 matrix when H/h was set to 50.

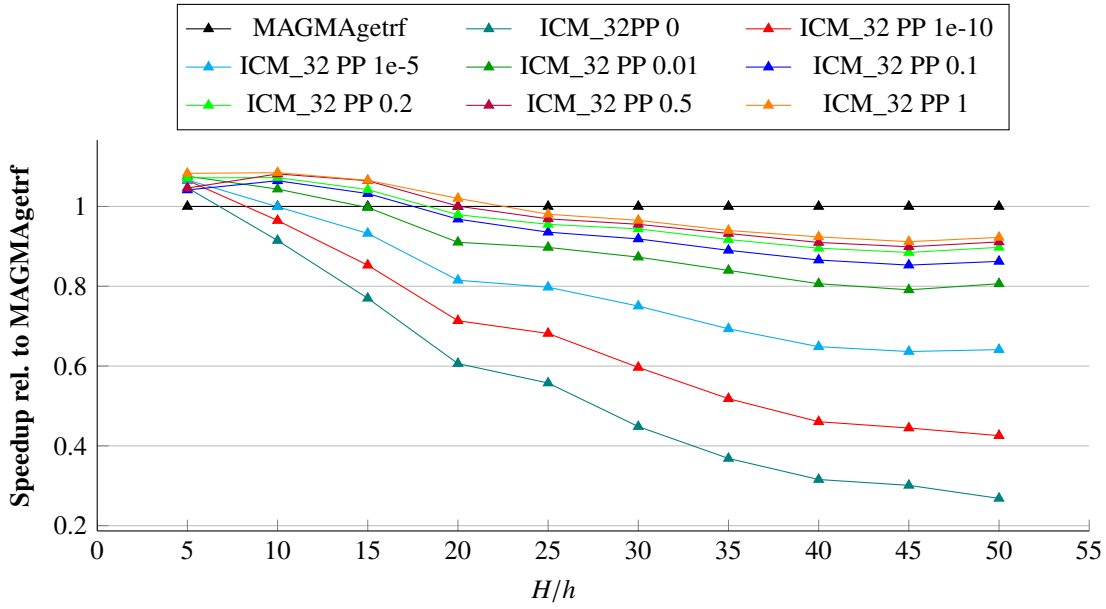


(d) Iterative and pivoting metrics of ICM_32PP collected during the decomposition of a 7,664-by-7,664 matrix when H/h was set to 50. Each square represents a 768-by-768 section of the matrix.

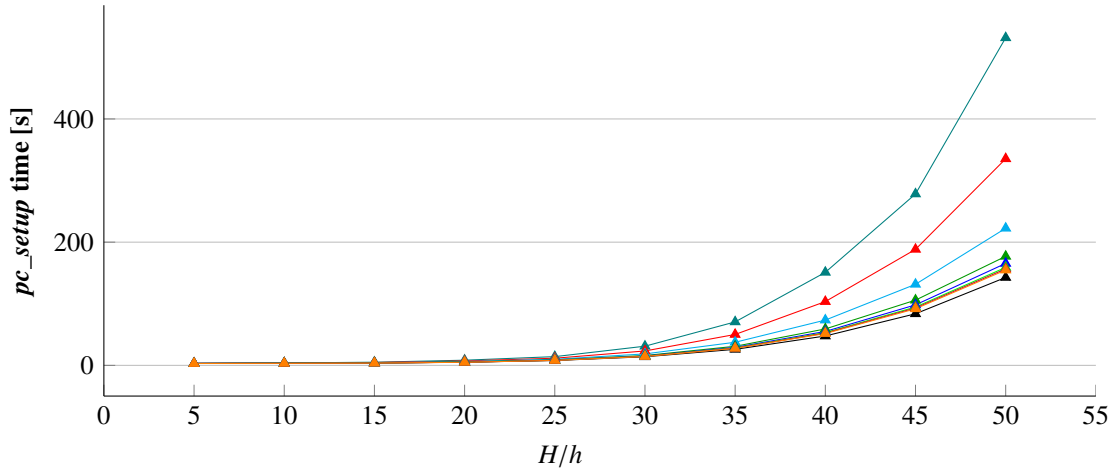
Figure 3.21: Visualization of the nonzero element patterns of LU matrices produced by ICM_32PP in the *pc_setup* stage for $H/h \in \{25, 50\}$. The figure includes the iterative and pivoting metrics of ICM_32PP. In the LU plot, the color of nonzero elements depends on their absolute value. Small entries are light orange, large entries are black, and the color of mid-range entries ranges from light green to deep blue depending on the median of $\log_{10}$ of the nonzero values (with slight alterations) [55]. The color of zero entries is white. In the metrics plot, the color in each square represents the number of iterations performed on that section. The left and top axes indicate the section ID in each dimension, and the bottom axis displays the number of pivots performed in a diagonal section.

3. Thirdly, despite the matrices being dense, the individual sections did not require many iterations to be processed. For example, approximately 86% of the elements in the input matrix for $H/h = 25$ were nonzero, while for $H/h = 50$, the percentage increased to 92%.

The summary of the decomposition benchmarks in Section 3.1.5 suggested that using a higher processing tolerance for ICM_*x*PP may be suitable when the decomposer is employed as a preconditioner that is not expected to be highly accurate. As mentioned earlier on Page 97, the *Krylov method* stage uses the `LU` matrices produced by the decomposer to iteratively improve the solution. Hence, an inaccurate decomposition could potentially benefit the overall execution time of the problem. To investigate this claim, ICM_32PP with various processing tolerances was utilized as a decomposer in the *pc_setup* stage. Only the ICM_32PP variant is presented since it consistently outperformed the other variants of ICM_*x*PP. Figure 3.22a illustrates the speedups of ICM_32PP with various processing tolerances relative to MAGMAgetrf. Furthermore, the figure also displays the execution times of the *pc_setup* stage for ICM_32PP with various processing tolerances and MAGMAgetrf.



(a) Speedup relative to MAGMAgetrf

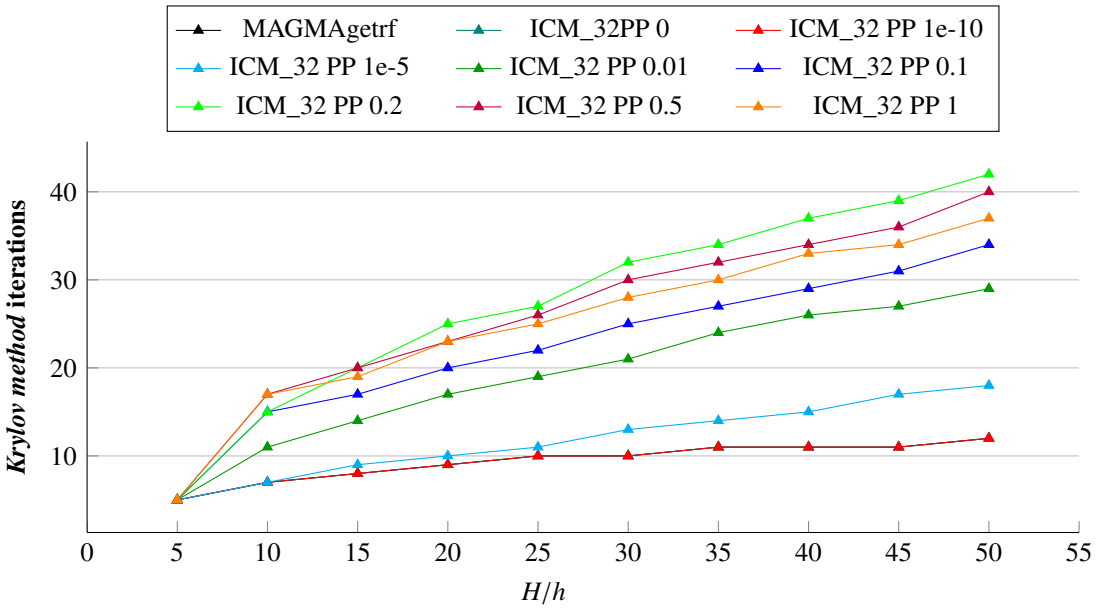


(b) Execution times

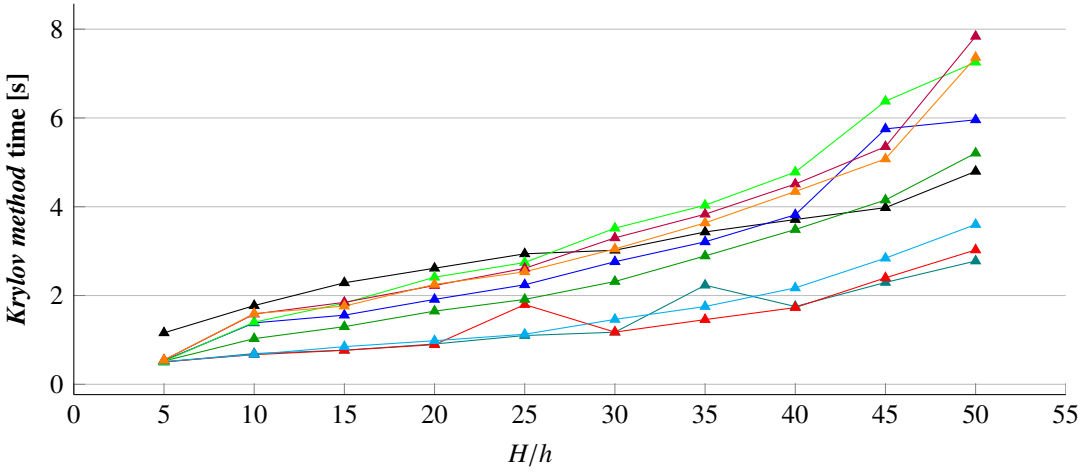
Figure 3.22: Speedup comparison of the *pc_setup* stage for ICM_32PP with various processing tolerances relative to MAGMAgetrf. The execution times of the stage are included for context. The name of each ICM_32PP decomposer follows the naming pattern *decomposer <processing_tolerance>*.

As depicted in Figure 3.22, increasing the processing tolerance of ICM_32PP resulted in improved performance. For example, with H/h set to 50, "ICM_32PP 1" achieved a speedup of 0.92 relative to MAGMAgetrf. It is worth noting that larger processing tolerances were not explored, as the performance gains between tolerances 0.5 and 1 were minimal, as shown in Figure 3.22a.

However, as described on Page 85, increasing the processing tolerance results in lower accuracy of the results. This decrease in accuracy is observed in the *Krylov method* stage, where the matrices produced by ICM_32PP are used. Specifically, a higher number of iterations are needed to achieve the desired accuracy of the solution, consequently leading to longer execution times of the stage. Figure 3.23 presents the number of iterations and the execution times of the *Krylov method* stage for ICM_32PP with various processing tolerances, as well as for MAGMAgetrf.



(a) Number of iterations



(b) Execution times (in seconds)

Figure 3.23: Number of iterations and execution times of the *Krylov method* stage for ICM_32PP with various processing tolerances and MAGMAgetrf. The name of each ICM_32PP decomposer follows the naming pattern *decomposer <processing_tolernace>*.

As expected, Figure 3.23a demonstrates that the number of iterations in the *Krylov method* stage increases as the processing tolerance values in ICM_32PP rise. However, as depicted in Figure 3.23b,

the corresponding execution times of the stage, while increasing, remain relatively small compared to those of the *pc_setup* stage. Consequently, the time saved by utilizing ICM_32PP with a high processing tolerance in the *pc_setup* stage far outweighs the time lost in the *Krylov method* stage due to the inaccurate results produced by the decomposer.

Comparison of the Highest-Performing Variants

For completeness, Figure 3.24 displays the speedup comparison of the best-performing variants, including the various processing tolerances, of each decomposer listed in Table 3.15, compared to MAGMAgetrf on the "Poisson equation on a cube" problem.

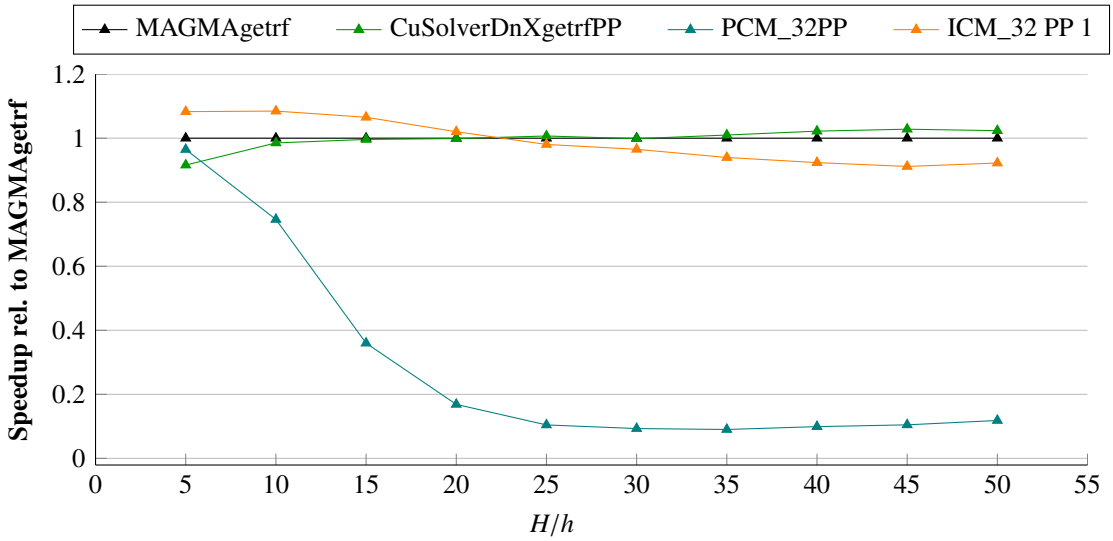


Figure 3.24: Speedup comparison of the best-performing variants, including the various processing tolerances, of each decomposer listed in Table 3.15, relative to MAGMAgetrf on the "Poisson equation on a cube" problem.

As depicted in Figure 3.24, overall, the "Poisson equation on a cube" problem was completed the fastest when CuSolverDnXgetrfPP was used along with CuBLAStrsmPP as the solver. Its performance was followed by MAGMAgetrf, and ICM_32PP 1 (with CuBLAStrsmPP as the solver).

Accuracy of Results

The accuracy of results in the BDDCML benchmark was evaluated based on the number of iterations required in the *Krylov method* stage. However, excluding ICM_*x*PP with increased processing tolerances, the number of iterations was the same for all decomposers across all values of H/h. In other words, the number of iterations for all decomposers listed in Table 3.15 matched that of MAGMAgetrf in Figure 3.23a.

3.3 TNL Time-Series Problem Benchmark

During the development of the project, a hypothesis was formulated by doc. Ing. T. Oberhuber, Ph.D., that the initial estimate in ICM_*x*PP could be beneficial for time-series problems. Specifically,

it was proposed that the LU matrix from the previous time step could be used as an initial estimate in ICM_*x*PP for the current time step, potentially reducing the number of iterations needed to process the matrix. This hypothesis was based on the observation that the input matrices undergo only slight changes between consecutive time steps, suggesting that the decomposed matrices may require fewer iterations to process.

For this purpose, doc. Ing. T. Oberhuber, Ph.D. implemented a benchmark for *Mean curvature flow*[16] in TNL[17]. The benchmark specifically focused on ICM_*x*PP and aimed to determine whether using the LU matrix from the previous time step as the initial estimate would improve performance compared to using the default initial estimate (the input matrix). Surprisingly, the benchmark results revealed no discernible difference in performance between the two approaches. Therefore, the benchmark results are not presented in this thesis. The reason behind the lack of difference remains unclear. However, it offers an interesting area for further investigation in future research.

3.4 Summary of Comparisons

This section summarizes the results of the benchmarks performed for the decomposers and solvers available in the Decomposition project.

Overall, the best-performing decomposer was CuSolverDnXgetrfPP, as it exhibited superior performance in both execution speed and accuracy in every benchmark it was a part of. When considering the decomposers implemented by the author of this thesis, PCM_*x*PP demonstrated consistent performance with an acceptable level of accuracy. However, for problems that require a preconditioner that does not need to produce highly accurate results, ICM_*x*PP is a suitable alternative. It is important to note that increasing the processing tolerance of ICM_*x*PP can lead to higher performance, however, there is no guarantee that it will produce usable results.

In terms of solvers, both CuSolverDnXgetrsPP and CuBLAStrsmPP were the best-performing, with the former being marginally faster than the latter while being slightly less accurate. It is worth noting that CuBLAStrsmPP allows for the unit diagonal to be stored either in L or U, whereas CuSolverDnXgetrsPP only supports a unit diagonal in L. While IS_*x*PP demonstrated that its iterative approach can avoid unnecessary calculations in certain problems, it produced invalid results for others, even with its processing tolerance set to zero. Therefore, it is not recommended for use.

[16] Mean curvature flow on Wikipedia: https://en.wikipedia.org/wiki/Mean_curvature_flow

[17] Mean curvature flow benchmark in TNL: https://gitlab.com/tnl-project/tnl/-/tree/TO/grid/src/Benchmarks/MeanCurvatureFlow

Conclusion

The objective of this thesis was to implement an iterative parallel LUP (Lower-Upper decomposition with Pivoting) algorithm, analyze its performance in BDDCML (multilevel Balancing Domain Decomposition by Constraints solver Library), and compare it to other established LUP implementations.

The first part of the thesis introduced the hardware of CPUs and GPUs, providing a basic understanding of their suitability for computations. The specifications of recent GPUs were also presented to strengthen the theoretical foundation. The relevant aspects of the software layer for code orchestration on Nvidia GPUs, CUDA, were then introduced, along with parallel computation concepts used in the implementation. Finally, the theoretical foundation concluded with the introduction of LUP algorithms and their application in solving systems of equations.

Then, the implementation of the project that encompassed procedures related to solving systems of equations using LUP was presented. Building upon the information presented in the first chapter, the implementations of the LUP procedures were detailed. Additionally, the unit tests, which assured the quality of the implemented algorithms, were presented, along with the benchmarks that were later used to evaluate the performance of the implementations.

Finally, the procedures implemented in the project were compared to established CUDA libraries in terms of execution speed and accuracy of results on a state-of-the-art HPC cluster. The comparison of performance was facilitated through two benchmarks. The first benchmark - Decomposition benchmark - evaluated the raw performance of the procedures on a set of 50 matrices with varying characteristics. The second benchmark - BDDCML benchmark - involved assessing the performance of the procedures as part of BDDCML.
The results of the Decomposition benchmark revealed that among the procedures implemented, the PCM_8PP (Parallel Crout's Method with Partial Pivoting and 8^2 threads in each one-dimensional CUDA thread block) decomposer exhibited the most consistent performance, while the ICM_32PP (Iterative Crout's Method with Partial Pivoting and 32-by-32 CUDA thread blocks) decomposer showed suitability only for specific types of matrices. However, both decomposers performed worse compared to the CusolverDnXgetrf procedure provided by CUDA. The implemented solver, IS_xPP (Iterative Solver with Partial Pivoting and x-by-8 CUDA thread blocks), was deemed unusable as it occasionally provided invalid results.
The second benchmark revealed that ICM_32PP was able to compete with CUDA's cusolverDnXgetrf and MAGMA's magma_dgetrf_gpu. Specifically, the relaxed processing tolerance of the iterative approach of ICM_32PP allowed it to complete the benchmark at 90% of the speed of magma_dgetrf_gpu.

As mentioned in Section 3.1.3, using the maximum absolute difference between actual and expected results as a measure of accuracy has its limitations. Furthermore, the time-consuming nature of the accuracy measurements restricted the number of matrices that could be included in the benchmark. Therefore, future work could include developing a more suitable and efficient approach

for measuring the accuracy of results. This revised approach could also help elucidate the issues highlighted in Sections 3.1.3 and 3.1.4. Further areas of future improvement could include refining the algorithm of ICM_*x*PP based on the benchmark results, and implementing project-based enhancements, such as automated benchmarks and automated processing of their results. These enhancements would significantly contribute to the overall efficiency and effectiveness of the project.

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. 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] *RCI Cluster Wiki* [online]. Czech Technical University, 2023 [visited on 2023-06-10]. Available from: https://login.rci.cvut.cz/wiki/start.

[53] DAVIS, T. A.; HU, Y. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* [online]. 2011, vol. 38, no. 1, pp. 1–25 [visited on 2023-06-10]. ISSN 0098-3500. Available from DOI: 10.1145/2049662.2049663.

[54] DAVIS, T. *SuiteSparse: A Suite of Sparse matrix packages at http://suitesparse.com: CSPY plot a matrix in color* [online]. [visited on 2023-06-16]. Available from: https://github.com/DrTimothyAldenDavis/SuiteSparse/blob/dev/CSparse/MATLAB/CSparse/cspy.m.

[55] DAVIS, T. A. *Direct Methods for Sparse Linear Systems* [online]. Philadelphia: Society for Industrial and Applied Mathematics, 2006 [visited on 2023-06-27]. ISBN 978-0-89871-613-9. Available from DOI: 10.1137/1.9780898718881.

[56] CHENG, J.; GROSSMAN, M.; MCKERCHER, T. *Professional CUDA C programming*. Indianapolis: John Wiley and Sons, 2014. ISBN 978-1-118-73932-7.

[57] DAVIS, T. A.; NATARAJAN, E. P. Algorithm 907. *ACM Transactions on Mathematical Software* [online]. 2010, vol. 37, no. 3, pp. 1–17 [visited on 2023-07-03]. ISSN 0098-3500. Available from DOI: 10.1145/1824801.1824814.

[58] AMESTOY, P. R.; DAVIS, T. A.; DUFF, I. S. An Approximate Minimum Degree Ordering Algorithm. *SIAM Journal on Matrix Analysis and Applications* [online]. 1996, vol. 17, no. 4, pp. 886–905 [visited on 2023-07-03]. ISSN 0895-4798. Available from DOI: 10.1137/S0895479894278952.

[59] AMESTOY, P. R.; DAVIS, T. A.; DUFF, I. S. Algorithm 837. *ACM Transactions on Mathematical Software* [online]. 2004, vol. 30, no. 3, pp. 381–388 [visited on 2023-07-03]. ISSN 0098-3500. Available from DOI: 10.1145/1024074.1024081.

[60] FROLOV, A. V. *Backward substitution* [online]. [visited on 2023-07-03]. Available from: https://algowiki-project.org/en/Backward_substitution.

[61] FROLOV, A. V. *Forward substitution* [online]. [visited on 2023-07-03]. Available from: https://algowiki-project.org/en/Forward_substitution.

[62] ŠÍSTEK, J.; OBERHUBER, T. Acceleration of a parallel BDDC solver by using graphics processing units on subdomains. *The International Journal of High Performance Computing Applications* [online]. [N.d.] [visited on 2023-06-11]. ISSN 1094-3420. Available from DOI: 10.1177/10943420221136873.

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

[64] 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.

[65] 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.

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

[67] 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 += 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 += 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 from the j-th row (incl. it) with max. value in the j-th
        column
56    Real max = TNL::abs( M( j, 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 Vector >
void
CroutMethod< BLOCK_SIZE >::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();

  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 );
      }
```

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

Listing B.1: Excerpt from the implementation of PCM_*x*PP. The template parameter `BLOCK_SIZE` is equivalent to x in PCM_*x*PP. 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 and 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;
18  }
```

Listing B.2: 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: 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 - lower bound
  const Real piv_tol_lower = 1.0e-5;
  // Pivoting tolerance - uppder bound
  const Real piv_tol_upper = 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
```

```
39  const Index nonDiagSecs_perRow = TNL::ceil( (double) num_cols / (double) ↩
        sec_size ) - 1;
40  auto* streams = (cudaStream_t*) malloc( nonDiagSecs_perRow * sizeof( ↩
        cudaStream_t ) );
41  for( Index i = 0; i < nonDiagSecs_perRow; ++i )
42    cudaStreamCreate( &( streams[ i ] ) );
43
44  // Flags indicating the processing status of non-diagonal sections
45  TNL::Containers::Array< bool, TNL::Devices::Cuda, Index > ↩
        nonDiagSec_processed{ nonDiagSecs_perRow, false };
46  TNL::Containers::Array< bool, TNL::Devices::Host, Index > ↩
        nonDiagSec_processed_host{ nonDiagSecs_perRow, false };
47
48  // Views are used from here on
49  auto A_view = A.getView();
50  auto LU_view = LU.getView();
51  auto LUnext_view = LUnext.getView();
52  auto piv_view = piv.getView();
53  auto processed_view = processed.getView();
54  auto nonDiagSec_processed_view = nonDiagSec_processed.getView();
55
56  // Lambda for fetching the index of bad elements
57  auto get_badEl_colIdxs = [ = ] __cuda_callable__( Index col ) -> Index
58  {
59    return ( abs( LU_view( col, col ) ) <= piv_tol_lower || piv_tol_upper < ↩
          abs( LU_view( col, col ) ) ) ? col : num_cols;
60  };
61
62  // Diagonal section start and end
63  Index dSec_start, dSec_end;
64  // Non-diagonal section start, end, and ID
65  Index sec_start, sec_end, sec_id;
66  // Row/Column index of the bad element
67  Index badEl_idx;
68
69  // Loop through the diagonal sections
70  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 ) )
71  {
72    // Set the starting point of the search for the first bad element
73    Index badEl_searchStart = dSec_start;
74
75    do { // Process the diagonal section and the sections below it
76      // Get next badEl_idx
77      badEl_idx = TNL::Algorithms::reduce< TNL::Devices::Cuda >( ↩
            badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, ↩
            num_cols );
78
79      do { // Process the diagonal section
80        // Reset the processed flag
81        processed_view.setElement( 0, true );
82
83        // Compute the values up to and including the bad element
84        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 );
85
86        // Assign the values computed for the next iteration
87        DSecAssign_kernel<<< blocks_perGrid, threads_perBlock >>>( LU_view, ↩
              LUnext_view, dSec_start, dSec_end, dSec_start, dSec_end, ↩
              badEl_idx );
```

```
88
89          // Check if a bad element is present in a different column
90          badEl_idx = TNL::Algorithms::reduce< TNL::Devices::Cuda >( ↩
                badEl_searchStart, dSec_end, get_badEl_colIdxs, TNL::Min{}, ↩
                num_cols );
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         // Launch kernels for all sections below the diagonal section - each ↩
               section has its own stream
109         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↩
                 )
110         {
111           // Only compute sections that are not yet processed
112           if( ! nonDiagSec_processed_host( sec_id ) ) {
113             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 );
114
115             NonDiagSecAssign_kernel<<< lSec_blockPerGrid, threads_perBlock, ↩
                    0, streams[ sec_id ] >>>( LU_view, LUnext_view, dSec_start, ↩
                    badEl_idx_cutoff, sec_start, sec_end );
116           }
117         }
118         // Wait until all sections have been computed in this iteration
119         synchronizeStreams( streams, nonDiagSecs_perRow );
120       } while( ! TNL::Algorithms::reduce( nonDiagSec_processed_view, TNL::↩
             LogicalAnd{} ) );
121
122       // Pivot the bad element
123       pivotBadElement< BLOCK_SIZE >( A_view, LU_view, piv_view, badEl_idx, ↩
             num_rows, num_cols, badEl_searchStart, piv_tol_lower, piv_tol_upper↩
              );
124
125     } while( badEl_idx < dSec_end - 1 );
126
127     nonDiagSec_processed_view.setValue( false );
128
129     // Process sections to the right of the diagonal section in parallel
130     // At this point there are no bad elements in the diagonal section
```

```
131     do {
132       nonDiagSec_processed_host = nonDiagSec_processed;
133       nonDiagSec_processed_view.setValue( true );
134
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 ICM_*x*PP. The template parameter `BLOCK_SIZE` is equivalent to x in ICM_*x*PP. 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_lower, const Real& piv_tol_upper )
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
6   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( isnan( max_elem.first ) ) {
    // The current bad element in the main diagonal is NaN -> Decomposition ↩
        failed
    throw Exceptions::NotANumber( j, j );
  }
  else if( max_elem.first <= piv_tol_lower || piv_tol_upper < max_elem.first ↩
      ) {
    // The value at the index of the bad element is not within the allowed ↩
        range, however, it is not zero/nan, 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
```

```
4 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 )
5 {
6   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  // Adjust 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: 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.

```
1  template< const int BLOCK_SIZE, typename ConstMatrixView, typename MatrixView↩
       , typename BoolArrayView, typename Index, typename Real >
2  __global__
3  void
4  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 )
5  {
6    Index tx = threadIdx.x;
7    Index ty = threadIdx.y;
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   // Adjust 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 right section, the row index is always smaller
21   Index min_row_col = row_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
```

```
  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;

  for( k = 0; k < ty; ++k )
    sum += L_block[ ty ][ k ] * U_block[ k ][ tx ];

  // Formula for U
  // Do not check for division by zero as this kernel assumes that no bad element is present in the diagonal section
  sum = ( A( row, col ) - sum ) / LU( row, row );

  // Check if the element (row, col) has been processed
  // Read LU( row, col ) from shared memory instead of global memory
  if( abs( U_block[ ty ][ tx ] - sum ) > process_tol )
    processed( sec_id ) = false;

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

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

```
template< typename MatrixView, typename Index >
__global__
void
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 )
{
  Index row = blockIdx.y * blockDim.y + threadIdx.y + sec_start_row;
  Index col = blockIdx.x * blockDim.x + threadIdx.x + sec_start_col;

  if( row >= sec_end_row || col >= sec_end_col )
    return;

  LU( row, col ) = LUnext( row, col );
}
```

Listing C.5: 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 += LU( i, j ) * Y( j );
      auto y_row = Y_view.getRow( j );

      auto rhsSum = [ = ] __cuda_callable__( Index rhs ) mutable
      {
        sum_view( rhs ) += LU_view( i, j ) * y_row.getValue( rhs );
      };

      TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, rhsSum );
```

```
43      }
44      // Y( i ) = ( B( i ) - sum ) / LU( i, i );
45      auto y_row = Y_view.getRow( i );
46      auto b_row = B_view.getRow( i );
47
48      auto ySet = [ = ] __cuda_callable__( Index rhs ) mutable
49      {
50        y_row.setValue( rhs, ( b_row.getValue( rhs ) - sum_view( rhs ) ) / ↩
              LU_view( i, i ) );
51      };
52
53      TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, ySet );
54    }
55
56    // Backward substitution: UX = Y
57    for( Index i = num_rows - 1; i >= 0; --i ) {
58      sum_view.setValue( 0 );
59
60      for( Index j = i + 1; j < num_cols; ++j ) {
61        // sum += LU( i, j ) * X( j );
62        auto x_row = X_view.getRow( j );
63
64        auto rhsSum = [ = ] __cuda_callable__( Index rhs ) mutable
65        {
66          sum_view( rhs ) += LU_view( i, j ) * x_row.getValue( rhs );
67        };
68
69        TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, rhsSum );
70      }
71      // X( i ) = Y( i ) - sum;
72      auto x_row = X_view.getRow( i );
73      auto y_row = Y_view.getRow( i );
74
75      auto xSet = [ = ] __cuda_callable__( Index rhs ) mutable
76      {
77        x_row.setValue( rhs, y_row.getValue( rhs ) - sum_view( rhs ) );
78      };
79
80      TNL::Algorithms::parallelFor< Device >( Index{ 0 }, nrhs, xSet );
81    }
82  }
```

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

IS_*x*PP Implementation

The implementation of IS_*x*PP 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();
  const Real process_tol = 0.0;
```

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

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
10   // Terminate threads that overreach matrix bounds
```

```
  if( row >= num_rows || rhs >= nrhs )
    return;

  Real sum = 0;

  for( Index j = 0; j < row; ++j )
    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: 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 += 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: 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 )
{
```

```
6   // Each thread assigns 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
10  // Terminate threads that overreach matrix bounds
11  if( row >= num_rows || rhs >= num_cols )
12    return;
13
14  M( row, rhs ) = Mnext( row, rhs );
15 }
```

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

Appendix F

Python Script for Generating Random Dense Matrices

The script used to generate random dense matrices for the benchmark implemented in the Decomposition project is presented in Listing F.1.

```
import numpy as np

from random import random

num_mtx_files = 15
num_rows_cols_range = [500, 11000]
elements_range = [-1000, 1000]
zeros_on_diag = True

mtx_files = np.random.randint(low=num_rows_cols_range[0], high=↩
    num_rows_cols_range[1], size=num_mtx_files)
pivoting = "_pivoting" if zeros_on_diag == True else ""

for num_rows_cols in mtx_files:
  dense_mtx_filename = f"Cejka{num_rows_cols}_<{elements_range[0]}-{↩
      elements_range[1]}>{pivoting}.mtx"
  print(f"Generating {dense_mtx_filename}...")

  header_lines = [
  "%%MatrixMarket matrix coordinate real general",
  "%-----------------------------------------------------------------------",
  "% Cejka Dense Matrix Collection",
  "% Insert URL here",
  f"% name: Cejka/{dense_mtx_filename}",
  "% date: 2023",
  "% author: L. M. Cejka",
  "% kind: Random Dense Matrix with 0.5 prob. of 0 on diag.",
  "%-----------------------------------------------------------------------",
  ]
  header_lines = [line + "\n" for line in header_lines]

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

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

```
35  zero_with_newline = str(0) + "\n"
36
37  num_zeros_on_diag = 0
38
39  with open(dense_mtx_filename, 'w') as f:
40    f.writelines(header_lines)
41    f.write(mtx_info)
42    for col in range(1, num_rows_cols + 1):
43      for row in range(1, num_rows_cols + 1):
44        if zeros_on_diag and row == col and row != num_rows_cols:
45          f.write(f"{row} {col} {rand_floats[(col - 1) * num_rows_cols + row
              - 1] if random() < 0.5 else zero_with_newline}")
46        else:
47          f.write(f"{row} {col} {rand_floats[(col - 1) * num_rows_cols + row
              - 1]}")
```

Listing F.1: Implementation of the `generate-rand-dense-mtxs.py` script used to generate dense matrices in the Matrix Market File Format, as described in Section 2.2.3. Each element in the matrix is randomly generated within the range of -1000 to 1000. The `zeros_on_diag` variable, declared on Line 8, serves as a flag indicating whether the dense matrix can have zeros on its main diagonal. When the variable is set to `True`, there is a 50% chance for each element on the main diagonal to be a zero, as shown on Line 45.