DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Linear Equation Solvers on GPU Architectures for Finite Element Methods in Structural Mechanics

Peter Wauligmann

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Linear Equation Solvers on GPU Architectures for Finite Element Methods in Structural Mechanics

Lineare Gleichungslöser auf GPU Architekturen für Finite-Elemente-Methoden in der Strukturmechanik

Author:	Peter Wauligmann
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisors:	Tobias Opel, M.Sc.
	Hayden Liu Weng, M.Sc. (hons)
Submission Date:	15.04.2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.04.2020 Peter Wauligmann

Abstract

Many scientific computing applications rely on linear equation solvers for numerical simulations. Moving away from traditional parallelization models, we investigate possibilities to accelerate linear equation solvers by using hybrid architectures consisting of CPUs and GPUs. The targeted equation systems originate from finite element method simulations in structural mechanics performed by *CalculiX*. We review existing iterative and direct solvers for shared memory and GPU parallelization. Iterative methods cannot compete with direct strategies for the given benchmark matrices because of stagnating convergence. Direct solvers are known to be memory demanding and thus a block-wise offloading scheme was identified as the most promising approach to utilize GPGPU.

Instead of developing a new solver we evaluate that *PaStiX* implements this strategy very efficiently and performs well for relevant matrices. An in-depth analysis is conducted to find bottlenecks and potential enhancements. A general weakness is the amount of sequential and unaccelerated code before and after the factorization. To reduce the effect of Amdahl's law, those sections are extended with OpenMP and CUDA. The PCI Express bus is the factorization's bottleneck. Two optimizations are presented to relieve it. Firstly, a mixed precision strategy that allows computation in `float` instead of `double` and thus halves the amount of data to be sent. In total, this method leads to 46% and 20% higher performance in pure CPU and hybrid mode. Secondly, the effects of using pinned memory are analyzed. On average, the enhanced bandwidth leads to 20% faster factorization but does not improve the total computation time since allocating pinned memory is costly. For most matrices, an absolute speedup is achieved when using the same memory range for 7 or more factorizations.

Finally, PaStiX is integrated into CalculiX and, in fact, reusing is not only possible for pinned memory buffers but also for reordering permutations. This way, the scalable numerical parts of PaStiX account for 66.1% instead of 26.8% of the computation time. Previously, CalculiX used the CPU-based library PARDISO. Depending on the input deck, the enhanced PaStiX solver runs between 2.6 and 12.7 times faster. Considering the entire CalculiX application, an average speedup of 3.1 for the CPU and 4.4 for the hybrid version is achieved based on the original implementation.

Contents

1 Introduction **1**

2 Fundamentals **3**

2.1 Structural Mechanics and the Finite Element Method . . . 3

2.1.1 Governing Equations . . . 3

2.1.2 Finite Element Method . . . 5

2.1.3 Solving Non-Linear Problems . . . 7

2.2 Sparse Matrices . . . 8

2.2.1 Coordinate Format (COO) . . . 9

2.2.2 Compressed Sparse Row Format (CSR) . . . 9

2.2.3 Compressed Sparse Column Format (CSC) . . . 9

2.3 Hardware Considerations . . . 10

2.3.1 Graphical Processing Units . . . 10

2.3.2 Amdahl's Law . . . 11

2.4 Direct Methods for Solving Systems of Equations . . . 12

2.4.1 Gaussian Elimination Algorithm and LU Decomposition . . . 13

2.4.2 Cholesky and LDLT Decomposition . . . 16

2.4.3 Sparsity and Reorderings . . . 17

2.4.4 Supernodal Methods and Trees . . . 21

2.5 Iterative Methods for Solving Systems of Equations . . . 22

2.5.1 Matrix Splitting . . . 22

2.5.2 Preconditioning . . . 23

2.5.3 Krylov Subspace Methods and Preconditioned GMRES . . . 24

3 Analysis and Related Work **26**

3.1 Hardware Configuration . . . 26

3.2 Problem Sets . . . 27

3.3 CalculiX . . . 27

3.3.1 CalculiX's Matrix Format . . . 29

3.4 Sparse Direct Solvers . . . 30

3.4.1 Accelerator Implementations . . . 31

3.4.2 Hybrid Implementations 32
3.5 Sparse Iterative Solvers 33
3.6 PaStiX 34

4 Mixed Precision in PaStiX **39**
4.1 Single and Double Precision 39
4.2 Half Precision 43

5 Optimizations for PaStiX **46**
5.1 Parameter Tuning 46
5.2 Pinned Memory 50
5.3 Parallel Matrix Preprocessing 52
5.3.1 Parallel Permutation of a Matrix in CSC Format 53
5.3.2 Transpose of a Structurally Symmetric Matrix 54
5.3.3 Performance Evaluation 56
5.4 GPU-Accelerated Iterative Refinement 57

6 PaStiX Integration in CalculiX **61**
6.1 Conversion of CalculiX's Matrix Format 61
6.2 Reusing Matrix Patterns 62
6.2.1 Reusiability Optimization 63
6.3 Evaluation 64
6.3.1 Amortized Performance of PaStiX 65
6.3.2 Total CalculiX Acceleration 67

7 Conclusion **69**
7.1 Future Work 70

List of Figures **71**

List of Tables **73**

Bibliography **74**

1 Introduction

Numerical simulation is the method behind many significant scientific and industrial achievements. Possible application areas are meteorology, biology and engineering. Most people are influenced by such simulations in their everyday life. In some cases, this influence is very direct and obvious. A weather forecast, for example, is based on numerical simulation and something that many people consult before planning outdoor activities. In other domains the application is more subtle and therefore harder to recognize. Designing a sophisticated car requires more than one type of simulation. Experts in structural mechanics simulate the deformation behavior and operational life span. Specialists in aerodynamics determine the optimal shape of the vehicle body by performing computational fluid dynamics simulations and finally thermodynamic engineers use simulations to adjust the heat flow inside the car's engine.

The characteristic and goals of simulations vary based on the field of application. One thing they have in common is that an analytical solution does not exist for most problems and thus numerical approximation is the best way to assess the physical situation without performing expensive experiments. Numerical algorithms work on a discretization of the continuous problem, which is also called a mesh in this context. In three-dimensional models those usually consist of cuboids, tetrahedrons or hexahedrons. The objects in computational simulations have to follow the laws of physics. These are expressed in partial differential equations (PDEs) and also depend on the application domain. Laws of physics are inherently continuous and so are the associated PDEs. Thus, based on the mesh discretization, the equations have to be discretized likewise. The most popular discretization methods can be categorized into finite difference, finite volume and finite element methods.

In this work we analyze structural mechanics simulations using finite element methods. These usually require, in contrast to other discretization schemes, solving multiple systems of equations. For typical geometries, one solving process requires the execution of around 2 to 200 TFLOPs. In this context, 1 TFLOP stands for 10^{12} floating point operations. This is usually the most compute intensive part of the whole simulation process. To handle this load in reasonable time, a system that is able to perform multiple TFLOPs per second is desirable. The traditional approach to reach high performance is to deploy a cluster system of multiple compute nodes that communicate with each other in order to perform a joint simulation. In this work, however, we avoid distributed memory parallelization and try to maximize the performance on a single node. To elevate the computational capacity of a node, the central processing unit (CPU) can be assisted by accelerator hardware. In this work, we analyze the acceleration of the solving step for structural mechanics simulations using a graphical processing unit (GPU).

Solving a linear system of equations can be done with direct or iterative methods. Both approaches and other important fundamentals are introduced in chapter 2. In chapter 3 we consider the hardware and benchmark data selected for this project and evaluate existing direct and iterative solvers. We identify the direct solver *PaStiX* as the most promising library for our purposes and subsequently develop a mixed precision strategy for it in chapter 4. Chapter 5 introduces further individual optimizations to the solver in order to prepare it for efficient and productive use in the simulation software *CalculiX*. The integration into CalculiX and tailored optimizations are described in chapter 6. Eventually, chapter 7 reviews the contributions of this thesis and discusses possible future work.

2 Fundamentals

To understand the essence of this work, certain knowledge from different areas is required. This chapter intends to assist the reader with an introduction into selected topics. For more in-depth background knowledge, the reader can consult the mentioned literature.

Section 2.1 introduces the governing equations for a basic example in structural mechanics. In section 2.2 sparse matrices and their most important storage schemes are presented. Section 2.3 covers GPUs and the caveats when programming them. Finally, the sections 2.4 and 2.5 introduce direct and iterative methods for solving linear systems of equations.

2.1 Structural Mechanics and the Finite Element Method

Simulations in structural mechanics are performed in order to predict how a real object performs under load. Depending on the problem formulation, the governing equations can be linear or non-linear. This section briefly introduces the physical problem, its discretization and how non-linearity is handled.

2.1.1 Governing Equations

The purpose of computational models in structural mechanics is to evaluate stresses and deformations based on input geometry and forces. Because general methods for the required techniques are too complex and not mandatory to understand the essentials of this work, we only provide a brief introduction based on the simple 1D axially loaded bar problem discussed in [Cha20] and visualized in figure 2.1.

The loaded bar model is parametrized with four variables:

1. The length of the bar L.
2. The cross section A.

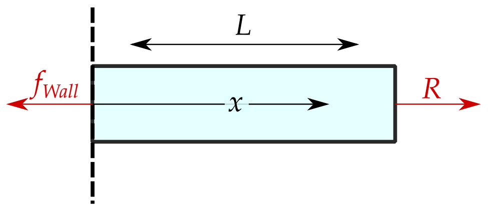

Figure 2.1: 1D axially loaded bar.

3. The loaded force R.
4. The Young's modulus E, which describes the material's stiffness.

Additionally we assume

- A constant cross section
- Linear elastic, isotropic, homogeneous material
- Centric load

Based on the balance of forces $f_{Wall} = f(x) = R$ the stresses σ are

$$\sigma(x) = \frac{f(x)}{A} = \frac{R}{A}.$$

Due to Hooke's Law, which states $\sigma = E\epsilon$, the strain ϵ can be expressed as

$$\epsilon(x) = \frac{\sigma(x)}{E} = \frac{R}{AE}.$$

Using the strain-displacement relations ($\epsilon(x) = \frac{\delta(x)}{x}$) the displacements inside the bar are given by

$$\delta(x) = \frac{Rx}{AE}.$$

Further, we need the equation in differential form, which is used in numerical methods. By satisfying the equilibrium equation $A\sigma = A(\sigma + \Delta\sigma)$, it follows that

$$A\frac{d\sigma}{dx} = 0 \Longrightarrow A\frac{dE\epsilon}{dx} = 0.$$

This time, we can apply the strain-displacement relation in its differential form ($\epsilon = \frac{du}{dx}$)

leading to the final differential equation

$$AE\frac{d^2u}{dx^2} = 0$$

and its boundary conditions

$$u(0) = 0$$

$$\sigma(L) = 0 \Longrightarrow AE\left.\frac{du}{dx}\right|_{x=L} = R.$$

Based on this differentially derived equation, called the *strong form*, an equivalent weak form of the equation can be arranged which is required for the finite element method. We will not cover that step in this work, but motivated readers might want to consult [Oña].

2.1.2 Finite Element Method

The finite element method (FEM) is a general protocol for approximating solutions of partial differential equations (PDEs). The main concept is to subdivide a geometry into many smaller elements that again consist of nodes. There are many different implementations of the finite element method but most of them feature the following characteristics:

1. Geometric Discretization: The continuous problem domain is transformed into a discrete geometry, which is then further divided into elements consisting of nodes.
2. Variational Method: Definition of basis and shape functions for element-wise interpolation based on node values.
3. Algebraical Equation Solver: Solving of a system of equations that was derived based on the PDEs, shape functions and initial values.
4. Post Processing: Error Analysis, possible adaptive refinement and highlighting of key data.

All of the characteristics are already widely researched and the state-of-the-art techniques for each of them have become very complex. To still provide an intuition for the reader, we show how the FEM can be applied on the 1D axially loaded bar. A more detailed explanation for this case is presented in [Ban07].

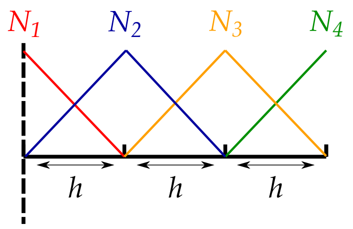

Figure 2.2: Node partitioning of the axially loaded bar, inspired by [Ban07].

First, the geometry is divided into three elements and four nodes, resulting in the mesh and basis functions N_i shown in figure 2.2. Based on the PDEs weak form

$$\int_0^L AE\frac{du}{dx}\frac{d(\delta u)}{dx}dx = R\delta u|_{x=L}$$

we restructure the system of equations, so that it has the form

$$Ku = f$$

where u are the unknowns, K the stiffness matrix and f the right hand side containing the loaded forces. K and f are defined as

$$K_{i,j} = \int_0^L AE\frac{dN_j}{dx}\frac{dN_i}{dx}dx$$

$$f_i = N_j\, R|_{x=L}$$

where N_i is defined as the basis function of a node i. Each entry $K_{i,j}$ represents an interpolation of values between node i and j. This relation is symmetric for linear models. If a node i does not have overlapping basis functions with a node j, the entry $K_{i,j}$ will be zero. For the example geometry the non-zero entries are $K_{1,1}$, $K_{1,2}$, $K_{2,1}$, $K_{2,2}$, $K_{2,3}$, $K_{3,2}$, $K_{3,3}$, $K_{3,4}$, $K_{4,3}$ and $K_{4,4}$. Due to the nature of the linear basis functions, $\frac{dN_i}{dx}$ is either $\frac{1}{h}$ or $\frac{-1}{h}$. Considering the boundary conditions

$$u(0) = 0 \qquad \text{and} \qquad AE\left.\frac{du}{dx}\right|_{x=L} = R.$$

the system of linear equations $Ku = f$ is

$$\frac{AE}{h}\begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ R \end{pmatrix}.$$

After obtaining the node-based displacements u, the element-based displacements and stresses can be computed by interpolating over the node-values once more using their basis functions.

It is interesting to note that the stiffness matrix in the example is identical to the one that is obtained using an order one central differences scheme of the finite difference method. Nevertheless, for more complex problems, the finite element method allows more flexibility and is therefore the most popular discretization strategy in structural mechanics.

2.1.3 Solving Non-Linear Problems

Non-linearity is naturally introduced by geometry and material [Rus15]. Moreover, situations in which two objects make contact lead to heavily non-linear behavior. According to [Rus15], contact is made in the following cases:

1. A body approaches a rigid surface and which results in its deformation.
2. Two bodies approach and subsequently deform each other.
3. Two separated zones of one body touch.

In most cases, non-linear equations cannot be solved directly and the iterative Newton-Raphson method is applied to solve such an equation $f(x) = 0$. The iteration formula is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

for one variable and

$$x_{i+1} = x_i - \left(\left. \frac{df(x)}{dx} \right|_{x=x_i} \right)^{-1} f(x_i) = x_i - {K_T}^{-1} f(x)$$

for multiple variables. K_T is the tangential stiffness matrix of the finite element method. Since determining the inverse of K_T is not feasible, the system

$$K_T \Delta x = -f$$

is solved for Δx every iteration where $\Delta x = x_{i+1} - x_i$. This solving step of a system of linear equations is the process we want to accelerate in this work. More information on non-linearity in structural mechanics can be found in [Rus15].

2.2 Sparse Matrices

Sparse matrices are matrices that contain many entries $a_{i,j} = 0$. They often emerge when dealing with discretized geometries. Those could be interpreted as graphs with coordinates. And just like adjacency matrices can represent a graph, we can arrange matrices to describe the influence of one discrete element on another. Since many elements only influence their direct neighbors, most of the matrix's entries are zero. Storing such matrices naively would require an immense amount of memory because the number of entries scales quadratically with the number of elements. Fortunately, only the non-zero entries have to be stored, together with a mapping that allows to identify each entry's row and column. Many different storing strategies exist and every application uses the one that is optimal for their requirements. They differ in memory consumption and the order in which the values are stored in memory. In this section we introduce several storage schemes that are relevant for this work. More formats can be found in [Saa03]. To facilitate the reader's comprehension we will provide conversions of the matrix in equation (2.1) for each of the presented storage schemes.

$$A = \begin{pmatrix} 1 & 0 & 4 & 0 & 0 & 6 \\ 0 & 0 & 0 & 9 & 0 & 0 \\ 5 & 0 & 2 & 0 & 0 & 0 \\ 0 & 7 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.1}$$

$$m = n = 6 \qquad\qquad nnz = 10$$

2.2.1 Coordinate Format (COO)

The coordinate representation of a sparse matrix is the most intuitive way to describe a sparse matrix. Each non-zero value is stored in combination with its row m and column n. The required memory is given by $3 \cdot nnz$.

Table 2.1: Matrix of equation (2.1) in coordinate format

`rows`	1	3	6	4	1	3	2	4	5	1
`cols`	1	1	1	2	3	3	4	4	5	6
`values`	1	5	8	7	4	2	9	3	10	6

2.2.2 Compressed Sparse Row Format (CSR)

Like the coordinate format, the compressed sparse row format contains the associated row for each value. However, instead of storing all the column indices, only offsets to the first value in each row are provided. The values have to be in row-major order. The required memory is given by $2 \cdot nnz + m + 1$.

Table 2.2: Matrix of equation (2.1) in CSR format

`rowptr`	1	4	5	7	9	10	11			
`cols`	1	3	6	4	1	3	2	4	5	6
`values`	1	4	6	9	5	2	7	3	10	8

2.2.3 Compressed Sparse Column Format (CSC)

The compressed sparse column format is closely related to the compressed sparse rows format. The values are stored in column-major order. Instead of pointers to the beginning of each row, it stores pointers to the beginning of each column. The required memory is given by $2 \cdot nnz + n + 1$.

Table 2.3: Matrix of equation (2.1) in CSC format

`rows`	1	3	6	4	1	3	2	4	5	6
`colptr`	1	4	5	7	9	10	11			
`values`	1	5	8	7	4	2	9	3	10	6

2.3 Hardware Considerations

2.3.1 Graphical Processing Units

Graphical processing units (GPUs) were invented in the late 90s [McC10]. Whilst early models contained specialized processing units for rendering and projections, later models introduced more versatile compute units that allowed general purpose computation. With the introduction of CUDA, a programming interface for NVIDIA devices, general purpose computing on graphical processing units (GPGPU) became popular [McC10]. The sheer number of compute units, called CUDA cores for NVIDIA devices, provides massive parallelism and performance. Due to the lack of individual scheduling units, multiple cores are forced to perform the same operation in each cycle. In Flynn's taxonomy, the parallelization model of modern GPUs is classified as "Single Instruction Multiple Data" (SIMD). In the past, there have been vector processor that followed the SIMD model but they have not been successful for long. GPUs, however, became an integral part of modern computer architectures for scientific computing.

The performance progression of CPUs has been described by Moore's law [Mac11] for over 50 years. It says that every two years, for the same cost, the number of transistors on a CPU chip doubles. The increase in clock frequency stagnated already around 2005 because heat production and energy consumption became too high. That is why the additional transistors were used to put multiple cores on one chip to increase the performance through multi-threading. Over the last decade, experts have predicted and finally announced that Moore's law is no longer valid [Wal16]. CPU manufacturers are not able to keep up the pace as transistors cannot become much smaller with the current technology. This progression favors GPUs because they can host far more computing units than CPUs. This has been the reason why modern GPUs perform more floating point operations per second (FLOP/s) than CPUs. Nvidia's current flagship card, Tesla V100, contains 2560 CUDA cores [NVI17] and a pricewise comparable Intel Xeon 8280 contains 28 cores with 2 compute units each [Cor19b]. Of course one has to consider that the CPU cores can perform vector instructions on 8 double precision values simultaneously at a much higher clock frequency. Nevertheless, the sheer number of CUDA cores outperform the CPU in theoretical peak performance. This performance advantage remains only if developers are able to write highly parallel applications, which is far more difficult considering the SIMD nature of GPUs. To support the platform developers who are facing this challenge, CPU and GPU vendors offer optimized libraries for standard algorithms. These can be BLAS libraries (Intel MKL, cuBLAS) or domain specialized solutions (cuDNN, MIOpen).

In a classical CPU+GPU architecture, one CPU is connected to one GPU via PCI Express [Wil13]. The GPU has its own memory and address range. The classical architecture can be extended by additional GPUs. Unfortunately, the PCI Express connection is not able to keep up with the main and device memory. Thus, it tends to become a performance bottleneck for applications that cannot avoid frequent communication between CPU and GPU.

Table 2.4: Comparison of different memory types [Shi19; Aja09; NVI17].

	Bandwidth in GB/s
DDR4	60+
PCI Express 3.0	16
PCI Express 4.0	32
GPU's HBM2	900

2.3.2 Amdahl's Law

Another important consideration when developing a GPU application concerns Amdahl's Law. A GPU can never fully replace a CPU because it lacks the ability to reasonably host an ordinary operation system. Therefore a hybrid implementation that benefits from the strength of each processing unit is necessary to achieve convincing performance. Ideally, CPU and GPU are both fully occupied during the execution of the application. Otherwise Amdahl's law becomes relevant:

$$S(f,p) = \frac{1}{f + \frac{1-f}{p}} \tag{2.2}$$

Amdahl's law (eq. 2.2) gives an upper bound on speedup S depending on the number of processors p and the fraction of sequential parts f in an application [HM08]. Given infinite resources and perfect parallelization ($\lim_{p\to\infty}$), Amdahl's law results in a speedup of $\frac{1}{f}$. The law is usually applied to shared or distributed memory parallelizations but can naturally also cover implementations for accelerators.

We consider an example in which a program is accelerated by a GPU. The CPU's theoretical peak performance of 1 TFLOP/s are complemented by the GPU's 7 TFLOP/s. We assume that only the numerical parts of the application can be parallelized and the initialization and finalization cannot. For the CPU version, the numerical part accounts for 16 seconds. The initialization and finalization both require 2 seconds, so that in total 80% of the program can be accelerated by the GPU. Applying Amdahl's law will let us determine an upper bound for the speedup of an hybrid implementation.

$$f = \frac{2+2}{20} = 0.2 \quad \wedge \quad p = \frac{1+7}{1} \quad \Rightarrow \quad S(0.2, 8) = \frac{1}{0.2 + \frac{0.8}{8}} = \frac{10}{3} = 3.3\overline{3} \tag{2.3}$$

Equation (2.3) shows that a perfect speedup of 8 for the computational part results in a total speedup of $\frac{10}{3} = 3.3\overline{3}$. This illustrates that even though the GPU offers tremendous performance, the overall speedup will be mediocre if the parts without acceleration are higher than a few percent. One of the main considerations during the design phase of HPC applications has to be the maximization of parallelizable and accelerateable procedures.

2.4 Direct Methods for Solving Systems of Equations

This section shortly reviews how to directly retrieve x from the linear equation

$$A \cdot x = b$$

where A is a nonsingular $n \times n$ matrix. x and b are vectors of size n. For a more detailed introduction into direct methods readers can consult [DER87].

The common direct methods for solving systems of equations make use of the fact that the equation

$$L \cdot U \cdot x = b$$

is easy to solve, when L and U are triangular matrices. By applying forward and backward substitution, a solution for $Ax = b$ is obtained. Forward substitution represents solving the equation

$$Ly = b.$$

Based on the solution y, backward substitution with

$$Ux = y$$

is performed. The choice of an efficient algorithm to obtain a triangular matrix decomposition depends on the system's properties. This is shown in table 2.5. Matrices A that are $m \times n$ with $m \neq n$ require other strategies, such as the QR decomposition.

Table 2.5: Most triangular matrix decomposition algorithms are specialized for certain matrix types.

Algorithm	Matrix Type
Cholesky	Symmetric Positive Definite
LDL^T	Symmetric
LU	Regular

2.4.1 Gaussian Elimination Algorithm and LU Decomposition

Even though the Gaussian elimination algorithm can be used for more than one purpose, this work will focus on solving systems of linear equations. The two-step algorithm first aims at eliminating the lower triangular entries of a matrix A. This is achieved by scaling individual rows and subtracting one row from another. Within this section, the algorithms are performed on generic 3×3 matrices to illustrate the algorithms' behavior.

We consider the system of linear equations

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \implies \left(\begin{array}{ccc|c} a_{1,1} & a_{1,2} & a_{1,3} & b_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & b_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & b_3 \end{array}\right).$$

During the execution of the algorithm we call the diagonal entries, which we subtract from lower triangular entries, *pivots*. Given that all entries are nonzero, the first pivot is $a_{1,1}$. To eliminate the lower triangular entry $a_{2,1}$, the first row is scaled so that $a_{1,1} = a_{2,1}$. Therefore, the so called multiplier is $l_{2,1} = a_{2,1}/a_{1,1}$. Analogously, for eliminating $a_{3,1}$ the multiplier is $l_{3,1} = a_{3,1}/a_{1,1}$. Then we subtract the scaled first row to obtain

$$\left(\begin{array}{ccc|c} a_{1,1} & a_{1,2} & a_{1,3} & b_1 \\ 0 & a_{2,2} - l_{2,1} \cdot a_{1,2} & a_{2,3} - l_{2,1} \cdot a_{1,3} & b_2 - l_{2,1} \cdot b_1 \\ 0 & a_{3,2} - l_{3,1} \cdot a_{1,2} & a_{3,3} - l_{3,1} \cdot a_{1,3} & b_3 - l_{3,1} \cdot b_1 \end{array}\right).$$

The tasks for pivot $a_{1,1}$ are now completed, and the step is repeated with the next pivot $a_{2,2}$. The multiplier becomes $l_{3,2} = \frac{a_{3,2} - l_{3,1} \cdot a_{1,2}}{a_{2,2} - l_{2,1} \cdot a_{1,2}}$ and then the scaled second row is subtracted from the third row.

$$\left(\begin{array}{ccc|c} a_{1,1} & a_{1,2} & a_{1,3} & b_1 \\ 0 & a_{2,2} - l_{2,1} \cdot a_{1,2} & a_{2,3} - l_{2,1} \cdot a_{1,3} & b_2 - l_{2,1} \cdot b_1 \\ 0 & 0 & a_{3,3} - l_{3,1} \cdot a_{1,3} - l_{3,2} \cdot (a_{2,3} - l_{2,1} \cdot a_{1,3}) & b_3 - l_{3,1} \cdot b_1 - l_{3,2} \cdot (b_2 - l_{2,1} \cdot b_1) \end{array}\right)$$

The resulting matrix is of triangular shape which enables us to peform simple backward substitution. The LU decomposition algorithm is very similar to the Gauss elimination. The only difference is that we store the multipliers in a matrix

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{2,1} & 1 & 0 \\ l_{3,1} & l_{3,2} & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ a_{2,1}/a_{1,1} & 1 & 0 \\ a_{3,1}/a_{1,1} & (a_{3,2} - l_{3,1} \cdot a_{1,2}) \,/\, (a_{2,2} - l_{2,1} \cdot a_{1,2}) & 1 \end{pmatrix}.$$

The upper triangular matrix is stored as

$$U = \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ 0 & u_{2,2} & u_{2,3} \\ 0 & 0 & u_{3,3} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a_{2,2} - l_{2,1} \cdot a_{1,2} & a_{2,3} - l_{2,1} \cdot a_{1,3} \\ 0 & 0 & a_{3,3} - l_{3,1} \cdot a_{1,3} - l_{3,2} \cdot (a_{2,3} - l_{2,1} \cdot a_{1,3}) \end{pmatrix}.$$

Blocked Algorithm

The common LU factorization does not allow the use of level 3 basic linear algebra subprograms (BLAS 3) which require the problem formulation in matrix-matrix operations. This can be done by applying a block Gaussian elimination or block LU decomposition [Huc17]. Furthermore, it enables coarse grained parallelism, which increases the parallel efficiency for shared, distributed or accelerator based implementations. The algorithm is self-recursive. In each recursion a decomposition of four blocks, as shown in equation (2.4), is considered.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{2,1} \\ 0 & U_{2,2} \end{pmatrix} \tag{2.4}$$

$$= \begin{pmatrix} L_{1,1} \cdot U_{1,1} & L_{1,1} \cdot U_{1,2} \\ L_{2,1} \cdot U_{1,1} & L_{2,1} \cdot U_{1,2} + L_{2,2} \cdot U_{2,2} \end{pmatrix} \tag{2.5}$$

The algorithm consists of five steps:

1. Subdivide A into four blocks.
2. (Recursively) Perform the LU algorithm on $A_{1,1}$.
3. Perform triangular solves on $L_{2,1} \cdot U_{1,1} = A_{2,1}$ and $L_{1,1} \cdot U_{1,2} = A_{2,1}$ to obtain $L_{2,1}$ and $U_{1,2}$.
4. Compute $L_{2,1} \cdot U_{1,2}$ and subtract it from $A_{2,2}$ to obtain $A_{2,2} = L_{2,2} \cdot U_{2,2}$.
5. (Recursively) Perform the LU algorithm on $A_{2,2}$.

In practice, the matrix is usually not split into four blocks. A finer blocking can be used to perform the algorithm in a partially loop-based approach which is visualized in figure figure 2.3.

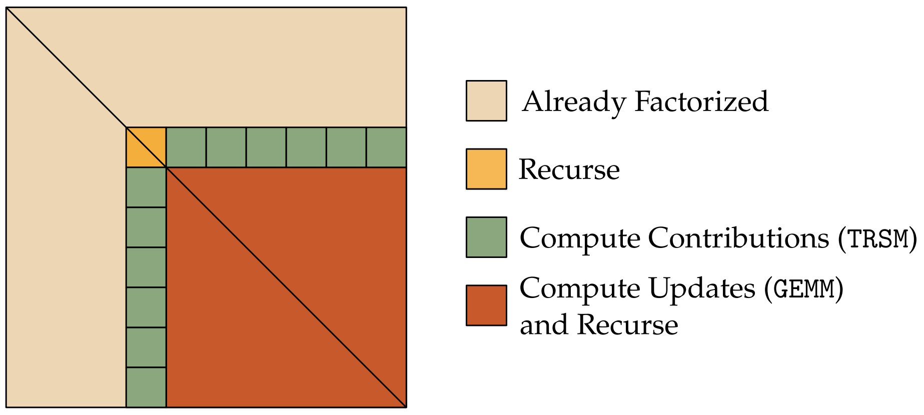
Already Factorized
Recurse
Compute Contributions (TRSM)
Compute Updates (GEMM) and Recurse

Figure 2.3: Visualization of block LU procedure.

This algorithm requires only two external functions to operate efficiently. Step 2 and 5 are merely self-recursive invocations until the blocks cannot be subdivided any further. To reduce the call stack one could consider a threshold block width, so that the recursion is replaced by a scalar LU algorithm. The BLAS 3 operation `TRSM` can perform the triangular solve for multiple columns, which is exactly the behavior required for step 3. In step 4, `GEMM` is applied to obtain the matrix product.

Performance Considerations

Before optimizing an implementation of the block LU factorization, it is important to evaluate the performance critical properties. The main question is whether the algorithm is compute- or memory-bound. A compute-bound problem is limited in performance because the hardware does not provide enough compute power, which we measure in FLOP/s. A memory-bound problem exhausts the bandwidth of the main memory while the processing units are not fully occupied.

In the previous paragraph, we evaluated that the block LU factorization largely consists of matrix multiplications (`GEMM`s) and triangular solves on matrices (`TRSM`s). Both operations are known to be compute bound, as their asymptotic computational intensity increases linearly with the problem size [PB12]. The computational intensity is measured in FLOPs per byte. A general matrix multiplication ($C = \alpha \cdot A \cdot B + \beta \cdot C$) for squared matrices of size n requires the transfer of $4 \cdot n^2$ values and computation of $2 \cdot n^3$ FLOPs. For double precision this implies

$$\text{Computational Intensity}(\texttt{GEMM}) = \frac{\text{FLOPs}}{\text{transferred bytes}} = \frac{2 \cdot n^3}{8 \cdot 4 \cdot n^2} = \frac{n}{16}.$$

`TRSM` performs triangular solves not just on a single vector (`TRSV`), but on a whole matrix ($L \cdot X = \alpha A$). It requires n^3 FLOPs and $\frac{5n^2+n}{2}$ memory transfers. These imply

$$\text{Computational Intensity}(\texttt{TRSM}) = \frac{\text{FLOPs}}{\text{transferred bytes}} = \frac{n^3}{4 \cdot (5n^2 + n)} \approx \frac{n}{20}.$$

In general we cannot assume squared blocks but the asymptotic values hold nonetheless.

We can use the arithmetic intensity to calculate the block size for which an operation transitions from memory- to compute-bound. Given that a computer provides 500 GFLOP/s in double precision and 50 GB/s memory bandwidth, problems with arithmetic intensity below 10 are memory-bound. For the mentioned characteristics the threshold is $n = 160$ for `GEMM` and $n = 200$ for `TRSM`. In the block LU algorithm, the block size will, due to the recursive domain decomposition, eventually become smaller than the threshold and thus a mixture of compute and memory bound operations is performed.

2.4.2 Cholesky and LDLT Decomposition

Instead of performing the decomposition

$$A = L \cdot U$$

one could store the diagonal explicitly in the form of

$$A = L \cdot D \cdot U.$$

For symmetric matrices, in which $a_{i,j} = a_{j,i}$, it can be deduced that $l_{i,j} = u_{j,i}$. and therefore $L = U^T$. This enables to rewrite the LDU factorization as

$$L = U^T \land A = LDU \implies A = LDL^T$$

This modification leads to the situation in which only half of the operations are required because the computation of the upper triangular matrix U can be omitted. The decomposition can be further simplified as into

$$A = LDL^T = (LD^{\frac{1}{2}})(D^{\frac{1}{2}}L) = \overline{L}\,\overline{L}^T.$$

This procedure is known as the Cholesky factorization. However, it is only applicable for symmetric positive definite matrices, because the expression $D^{\frac{1}{2}}$ requires computing the square root. If the matrix is not symmetric positive definite, the diagonal values might be negative.

2.4.3 Sparsity and Reorderings

The previous sections consider only dense matrices. Sparse matrices, as they appear in most numerical simulations, require additional considerations to work efficiently and as intended. One could think that sparse `GEMM` and `TRSM` operations could be applied to create a sparse version of the dense block LU algorithm, presented in section 2.4.1. The problem is that the LU factorization of a sparse matrix A is not necessarily as sparse as A.

$$\overset{A}{\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}} = \overset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 \\ 1 & -1 & 2 & 1 & 0 \\ 1 & -1 & 2 & \frac{2}{3} & 1 \end{pmatrix}} \cdot \overset{U}{\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 & -1 \\ 0 & 0 & -1 & -2 & -2 \\ 0 & 0 & 0 & 3 & 2 \\ 0 & 0 & 0 & 0 & \frac{5}{3} \end{pmatrix}} \tag{2.6}$$

$$nnz(A) = 13 \quad \land \quad nnz(L+U) = 25 \quad \Rightarrow \quad fillIn(A, L+U) = \frac{25}{13}$$

The exemplary matrix in equation (2.6) shows how a matrix, in which more than half of the entries are zero, becomes a dense matrix when performing LU factorization. In this context, fill-in describes the ratio between entries in the factorized and original matrix. To minimize the fill-in, row and column permutations can be applied. This process is called reordering. When the sparsity pattern of a matrix is symmetric, we want to preserve that property and restrict the reordering to symmetric permutations in which rows and columns are always permuted simultaneously. The reordering is expressed in the permutation matrix P so that the modified matrix can be expressed as PAP^T.

For the example presented in equation (2.6), a simple reordering that swaps rows and columns 1 and 5 greatly improves the situation, as shown in equation (2.7).

$$P = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \Longrightarrow PAP^t = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{2.7}$$

After performing the reordering on the example matrix, the LU factorization is already almost complete. Only the triangular matrices have to be extracted, which leads to less required operations and a fill-in of 1. This is a very artificial example but it shows how crucial a sophisticated reordering can be for sparse matrix factorization.

In this section we will briefly review two kinds of graph-based reordering algorithms and apply them the matrix and graph visualized in figure 2.4. The graph format has the advantage that partitioning algorithms can be performed more trivially.

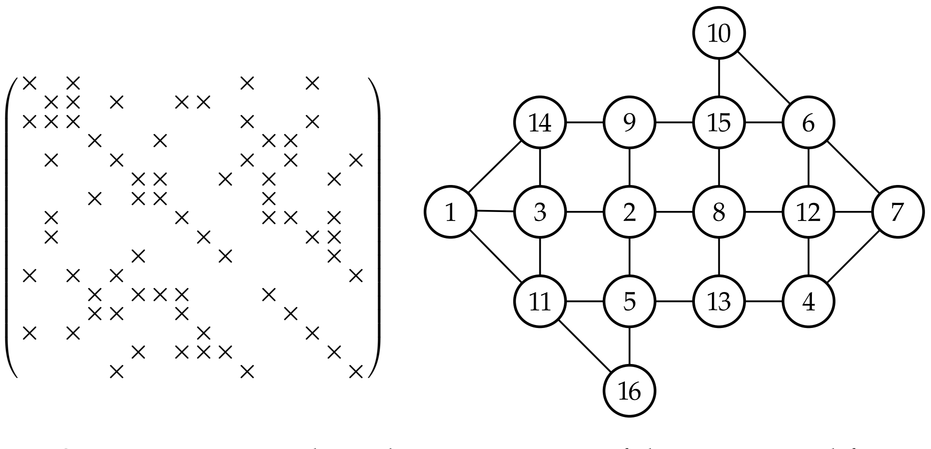

Figure 2.4: Sparsity pattern and graph representation of the matrix used for presenting the algorithms in this section.

Bandwidth Minimizing Algorithms

A banded matrix is a sparse matrix where $a_{i,j} = 0$ for $|i - j| > m$. m describes the semi-bandwidth of the banded matrix and can be interpreted as the farthest distance from an off-diagonal element to its closest diagonal element. The bandwidth of a matrix is then given as $2m + 1$. The advantage of banded matrices is that the Gaussian elimination only produces fill-in within the band structure. A common algorithm to obtain a reordering, that transforms an unstructured matrix into a banded matrix, is the reverse Cuthill McKee algorithm (RCM) [DER87]. The transformation of the sample matrix pattern from figure 2.4 is provided in figure 2.5. The banded structure is clearly recognizable and the bandwidth is $m = 5$.

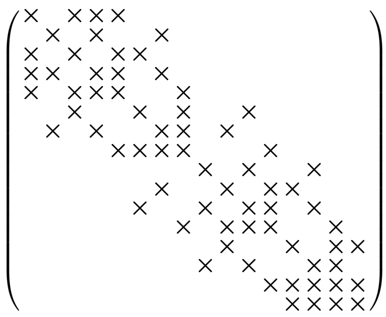

Figure 2.5: Sparsity pattern of figure 2.4 reordered with the RCM algorithm

Dissection Based Algorithms

The bandwidth minimizing approach presented in the previous section reduces the fill-in but does not support massive parallelization. The block LU factorization introduced in section 2.4.1 does allow to perform `TRSM` and `GEMM` in parallel but only for a single pivot block. The diagonal blocks always have to be performed in order. Especially for a banded matrix, the solve and update tasks are either not enough to allow fine-grained parallelism or too small to enable high arithmetic intensity.

An ordering is required that allows factorizing multiple diagonal blocks in parallel. Such ordering techniques are based on the (nested) dissection graph algorithms. The goal is to permute the entries so that we obtain a matrix of the form

$$PAP^T = \begin{pmatrix} A_{1,1} & 0 & A_{1,3} \\ 0 & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}. \tag{2.8}$$

Using this strategy, the factorization of blocks $A_{1,1}$ and $A_{2,2}$ can be executed completely independently and therefore in parallel. The nested dissection algorithm splits the graph representation into three partitions. Two of those should be as large as possible but are not allowed to contain any direct neighbor-vertices. To achieve that, the third partition, which should be as small as possible, hosts the vertices that act as separators. These vertex separators appear in our dissected matrix from equation (2.8) as $A_{1,3}$, $A_{2,3}$, $A_{3,1}$, $A_{3,2}$ and $A_{3,3}$. $A_{1,1}$ and $A_{2,2}$ represent the two partitions that were split from each other.

The graph in figure 2.4 requires at least three vertex separators to achieve a balanced partitioning, as shown in figure 2.6. The resulting sparsity pattern, displayed in figure 2.7, shows the typical structure of a matrix reordered by a dissection algorithm. This enables simple application of the block LU algorithm because the matrix is partitioned into blocks already.

The algorithm is called *nested* because like the block LU factorization, it is usually applied on the diagonal blocks recursively. The right graph of figure 2.4 shows the result of a second level recursion on our sample matrix.

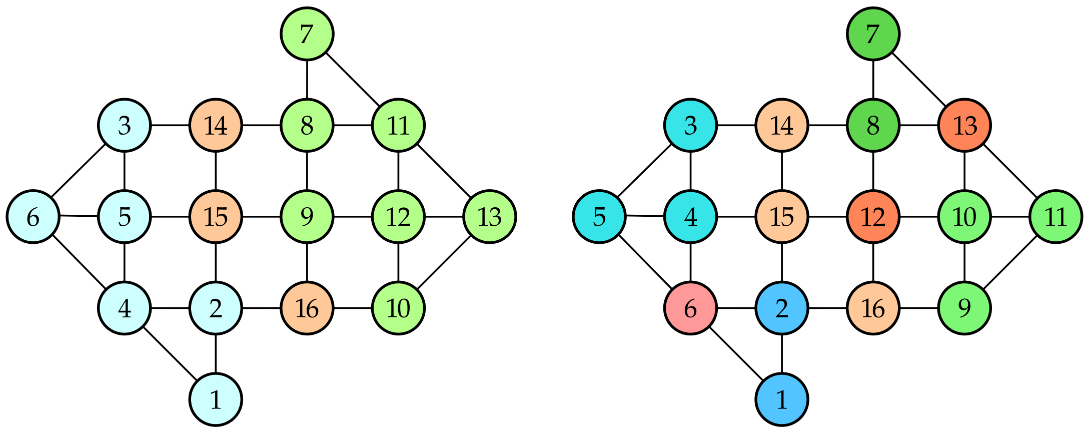

Figure 2.6: Graph partitioning for nested dissection applied to the graph of figure 2.4. The right graph is the result of one additional recursion. Redish nodes represent vertex separators.

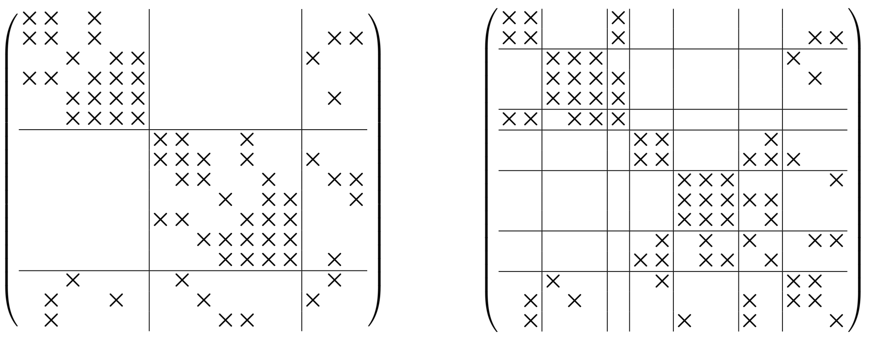

Figure 2.7: Sparsity pattern of figure 2.4 reordered with the nested dissection algorithm. The right matrix is the result of one additional recursion.

A common observation is that the vertex separators become smaller in deeper levels of recursion. This leads to the situation in which the blocks become small and the arithmetic intensity low. To avoid such situations, the recursion is terminated prematurely and a bandwidth minimizing algorithm like RCM is applied.

2.4.4 Supernodal Methods and Trees

In the previous section we discussed how to minimize the fill-in and how to maximize the parallelism by finding a suited reordering. In this section we introduce further techniques for the data structure of a direct solver and a more strategic approach for parallelization.

L and U cannot be stored in the same data structure as A because the fill-in alters the sparsity pattern and the factorized matrix is much denser than the original. Moreover, to use classical BLAS operations the values should not be stored in CSC or CSR but as a dense matrix. This is why the non-zero blocks retrieved by the nested dissection algorithm are stored individually as dense matrices. Since those block also contain zeros, we give up sparsity for a better-performing storage format. Due to the fill-in, many of the zeros that are stored explicitly become non-zero later anyways. The data structure is still partially sparse because many of the blocks contain exclusively zeros and are therefore omitted. In this context, the variables of each diagonal block and their contributions to other rows and columns are considered supernodes.

The nested dissection can also be interpreted as a tree data structure, which is then called the elimination tree [DER87]. It stores the vertex separators of each recursive invocation and the remaining vertices of the final recursion level as nodes. The elimination tree for the right matrix of figure 2.7 is visualized in figure 2.8.

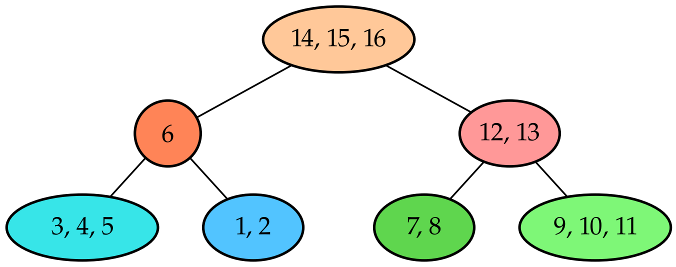

Figure 2.8: Elimination tree generated for the right matrix of figure 2.7.

The advantage of this data structure is that it already provides the dependencies between supernodes. To perform the factorization, `TRSMs` and `GEMMs` associated to a supernode, all of the node's children have to be processed first. The tasks for each of the leaf supernodes are independent and can be performed in parallel. For the sample matrix, this applies to the variables 1 to 5 and 7 to 11. Once the updates for variables 7 to 11 are completed, the factorization for the parent supernode (12, 13) can begin.

With these techniques we can compute large factorizations efficiently in parallel using dense BLAS 3 operations.

2.5 Iterative Methods for Solving Systems of Equations

Direct solving methods have several disadvantages, such as fill-in, required storage and the lack of simple parallelization approaches. Iterative methods intend to avoid these issues by keeping the matrix A unchanged during computation. The goal is still to solve $Ax = b$, although in iterative methods this solution is obtained by improving the error of a starting vector x_0 iteratively. The process $x_{k+1} := \Phi(x_k)$ is driven by the iteration function Φ, which is chosen such that $\lim_{k\to\infty} Ax_k - b = 0$ [Huc17]. The most fundamental work in this field is [Saa03] and should be considered for a thorough introduction, as this work merely provides a brief overview.

2.5.1 Matrix Splitting

The most basic approach for iterative methods is the splitting method in which the matrix A is split into two matrices $M - N = A$. The iteration step can then be written as

$$Mx_{k+1} = Nx_k + b = (M - A)x_k + b.$$

A splitting-based process, named Jacobi iteration, uses $M = D$ and $N = -(L + U)$, where D are the diagonal entries and L and U the upper and lower triangular submatrices [Saa03]. This leads to the iteration step

$$Dx_{k+1} = -(L + U)x_k + b. \Longrightarrow x_{k+1} = -D^{-1}(L + U)x_k + D^{-1}b.$$

Because $L + U = A - D$, we can rewrite that equation as

$$x_{k+1} = -D^{-1}(A - D)x_k + D^{-1}b = x_k + D^{-1}(b - Ax_k) = x_k + D^{-1}r_k$$

where $r_k = b - Ax_k$ is the residual for a solution x_k. The Jacobi method is only convergent for diagonal dominant matrices and known to be slow in convergence [Saa03]. Nevertheless, it is very easy to implement and parallelize for modern computer architectures, because only common BLAS operations such as matrix-vector and vector-vector multiplications are required. Other iteration processes such as the Gauss-Seidel method are faster in convergence but more difficult to parallelize.

2.5.2 Preconditioning

The basic iterative methods are, compared to direct solvers, efficient in computation and cheap in storage but often converge only slowly for matrices that deviate heavily from the identity matrix. A preconditioner $M \approx A$ can be used to transform the matrix so that

$$M^{-1}Ax = M^{-1}b$$

has to be solved instead of the original system. It is important that an approximate transformation $M^{-1}A \approx I$ can be found efficiently. Otherwise computing the preconditioner M^{-1} might take longer than is gained by faster convergence.

One example is the Jacobi preconditioner for which $M = D$, as introduced in the previous section. The result is a scaled matrix $D^{-1}A$ in which all diagonal entries are 1. For most cases, Jacobi preconditioning does not improve convergence significantly unless they are just poorly scaled[Wat15]. A more general preconditioner is the incomplete LU method (ILU) in which an approximate LU factorization $M = LU + \epsilon$ is performed. To avoid fill-in, only the non-zero entries in the matrix are processed during the factorization. Of course, this leads to a significant error ϵ but assures that the preconditioner is fast and memory-saving. Sometimes ϵ is too large and the preconditioning does not improve the convergence behavior of the iterative method. In such cases, a certain level of fill p can be allowed during the factorization, resulting in so called ILU(p) preconditioners.

Since the inverse of $M = LU$ is not known, the application of an ILU preconditioner has to be performed via forward and backward substitution, as introduced in section 2.4.

2.5.3 Krylov Subspace Methods and Preconditioned GMRES

Most matrices in large applications are not necessarily diagonally dominant and therefore, the splitting methods are only rarely used. Krylov subspace methods, such as GMRES and BiCGStab, are the most popular algorithms for asymmetric indefinite matrices since their introduction in the early 1990s. Because the mathematical foundation is complex and not directly related to the contribution of this work, we do cover mathematical theory but focus on the algorithmic aspect.

A Krylov subspace $K_m(A, r)$ is spanned by linear combinations of the basis $\{r, Ar, ..., A^r\}$ where r is the residual $b - Ax$ [Saa03]. Many algorithms based on the Krylov subspace exist. We introduce GMRES with right preconditioning, given in listing 2.1. The generalized minimal residual method (GMRES) minimizes the residual norm over $x_0 + K_m$. For the right-preconditioned algorithm, the basis of the Krylov subspace K is $\{r_0, AM^{-1}r_0, ..., (AM^{-1})^{m-1}r_0\}$.

```
$r = b - Ax_0$
$\beta = \|r\|_2$
$v_1 = r/\beta$
for $j = 1$ to $m$:
    $w = AM^{-1}v_j$
    for $i = 1$ to $j$:
        $h_{i,j} = (w, v_i)$
        $w = w - h_{i-j}v_i$
    $h_{j+1,j} = \|w\|_2$
    $v_{j+1} = w/h_{j+1,j}$
    $V_m = [v_1, ..., v_m]$
    $H_m = \{h_{i,j}\}_{1\leq i\leq j+1, 1\leq j\leq m}$
$y_m = \text{argmin}\|\beta e_1 - H_m y\|$
$x_m = x_0 + M^{-1}V_m y_m$
```

Algorithm 2.1: GMRES with right preconditioning [Saa03].

Initially, the residual r, its norm β and the first vector v_1 of the Krylov subspace's orthonormal basis V are computed. The preconditioner M^{-1} is applied in every iteration. Subsequently, starting from line 6, an Arnoldi process is applied to obtain the remaining v_j and the associated Hessenberg matrix H. Then, the residual can be minimized in the form of the least squares problem

$$\|b - Ax\|_2 = \|b - A(x_0 + V_m y)\|_2 = \|\beta e_1 - H_m y\|_2.$$

Finally, the preconditioner can be applied to the linear combination $V_m y_m$ and the result x_m is retrieved.

Further analysis with regards to computational considerations will be discussed in section 5.4.

3 Analysis and Related Work

In this chapter we analyze the problem of solving linear equations with regards to our hardware configuration and the application in non-linear structural mechanics. Therefore, section 3.1 introduces the computer architecture on which the benchmarks for this project are executed. In section 3.2 the problem sets that are generated by CalculiX and used as input for performance benchmarks are presented. In section 3.3 CalculiX, it's exotic sparse matrix format and the role of the equation solver are discussed. Existing implementations of direct and iterative solvers are analyzed in section 3.4 and section 3.5. In section 3.6 the direct hybrid solver PaStiX is evaluated thoroughly with respect to the hardware configuration and the problem sets.

3.1 Hardware Configuration

As part of this thesis, many benchmarks are performed to judge the effectiveness of applied optimizations. To obtain comparable results every benchmark is performed on the same hardware configuration. We refer to the computer as "T1". The CPU is an Intel Xeon Gold 6244 of the Cascade Lake generation released in 2019 [Cor19a]. It has 8 cores and provides a high frequency between 3.6 GHz and 4.4 GHz with Intel Turbo Boost. The equipped GPU is an NVIDIA Tesla V100 [NVI17]. Although it was already released in 2017, it is still the company's flagship card. It comes with 32 GB of high bandwidth memory (HBM2) and 2560 FP64 (double precision) compute cores. With a frequency of 1.53 GHz their accumulated performance is 7.8 TFLOP/s. In single (FP32) and half precision (FP16) they perform two and four times more operations. The V100 has additional compute units called tensor cores. They are restricted to half precision and mainly benefit machine learning applications, which have less demands to accuracy. Nevertheless, they elevate the performance of FP16 computations to 125 TFLOP/s.

The connection between GPU and main memory is PCI Express 3.0 with a maximum bandwidth of 15.75 GB/s [Aja09]. The main memory itself is DDR4 and transfers data to the CPU on 6 channels at in total 141 GB/s [Shi19].

When not otherwise mentioned, benchmarks are performed on T1 with 8 threads and GPU support.

3.2 Problem Sets

In this work we consider geometries that represent parts of a jet turbine, such as casings and blades. As the geometries have different properties, the matrices generated by CalculiX show distinctive characteristics. Those of the first iteration of each input deck are shown in table 3.1. We are performing benchmarks based on these nine inputs because they represent different characteristics that are common in structural mechanics. They are chosen based on their characteristics:

- Sample 1 and 3 are symmetric matrices because they do not contain contact elements. Sample 1 is, in contrast to sample 3, indefinite because it features ties. For these matrices, the LDLT algorithm can be applied.
- The simulation of some parts require finer geometries than others and therefore the resulting matrices are larger than others. The size is an important variable because it is the main influence for required memory, which is limited on GPUs. Larger geometries also tend to require more operations to solve. Therefore, they might allow more parallelism.
- The density of matrices depends on how the geometry is constructed. A denser matrix tends to introduce more fill-in and number of required operations for direct methods.

Furthermore, we will refer to the input decks as J1 to J9 and their first iteration's matrices as M1 to M9. The matrices are representative because they do not change fundamentally during the application of CalculiX.

3.3 CalculiX

CalculiX is an open source software for finite element simulations [DW98; Dho17]. It supports a wide range of analysis types, including static, dynamic, frequency and heat transfer. The FEM software is able to handle linear and non-linear behavior. CalculiX mirrors the interface of the commercial software *abaqus* to increase the usability for existing input decks. As introduced in section 2.1, FEM simulations require solving

Table 3.1: Matrices used for benchmarks in this work.

Matrix	Type	n	nnz	$density$
M1	Symmetric	$1.260 \cdot 10^6$	$1.333 \cdot 10^8$	$8.396 \cdot 10^{-5}$
M2	Asymmetric	$0.779 \cdot 10^6$	$1.203 \cdot 10^8$	$19.824 \cdot 10^{-5}$
M3	Symmetric PD	$4.983 \cdot 10^6$	$4.069 \cdot 10^8$	$1.639 \cdot 10^{-5}$
M4	Asymmetric	$1.898 \cdot 10^6$	$1.430 \cdot 10^8$	$3.970 \cdot 10^{-5}$
M5	Asymmetric	$2.263 \cdot 10^6$	$3.536 \cdot 10^8$	$6.905 \cdot 10^{-5}$
M6	Asymmetric	$3.847 \cdot 10^6$	$4.355 \cdot 10^8$	$2.943 \cdot 10^{-5}$
M7	Asymmetric	$2.222 \cdot 10^6$	$1.963 \cdot 10^8$	$3.976 \cdot 10^{-5}$
M8	Asymmetric	$2.053 \cdot 10^6$	$2.617 \cdot 10^8$	$6.209 \cdot 10^{-5}$
M9	Asymmetric	$1.231 \cdot 10^6$	$2.066 \cdot 10^8$	$13.634 \cdot 10^{-5}$

systems of linear equations. Since solving equations is a complex and much researched topic, CalculiX delegates this task to a specialized third party library.

Normally, one of the following libraries is used:

- SPOOLES, an open source solver developed in 1999 [AG99].
- PARDISO, a closed source solver that exists in two versions [SG04]. One is maintained by the original developer while the other one is distributed as part of Intel's MKL. In this paper we consider Intel's instance of the application and are referring to that when mentioning PARDISO.

In general, PARDISO is known to be faster than SPOOLES and many other direct solvers [SHG04]. Therefore, we will use CalculiX with PARDISO as the baseline for this work.

CalculiX implements shared memory parallelization with POSIX threads. To figure out the impact of the solver on CalculiX, we perform a profiling. As shown in figure 3.1, 59.1% or more of CalculiX's execution time is spent on solving systems of linear equations. CalculiX uses an internal sparsity format that is introduced in the next section. PARDISO requires the input matrix in CSR, which makes it necessary to apply a conversion algorithm. This takes between 0.18% and 13.73% of the total computation time. The conversions for J1 and J3 are very fast because they are in symmetric form, for which the storage formats barely differ.

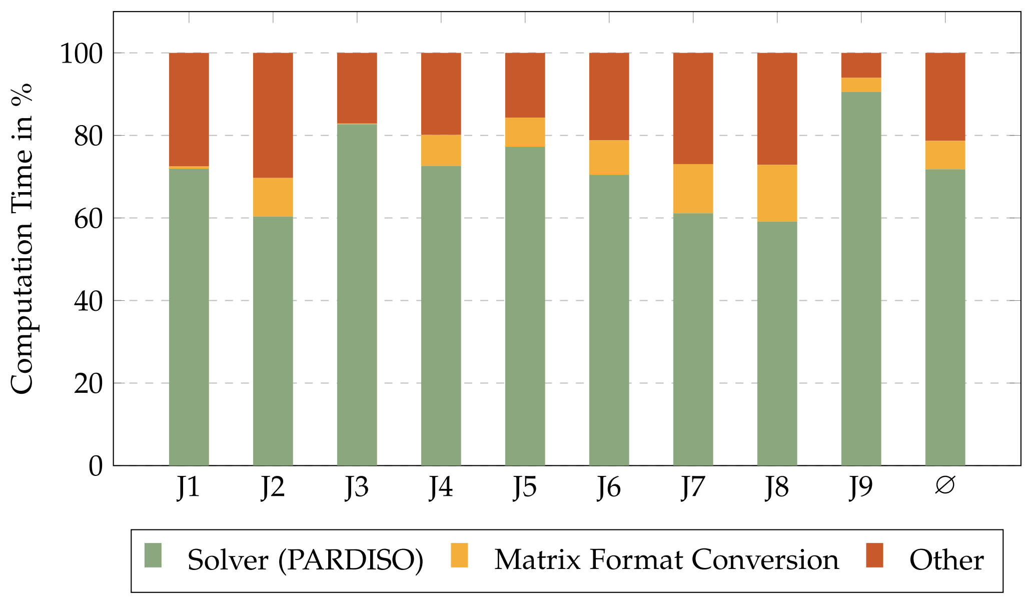

Figure 3.1: Structure of CalculiX's computation time with PARDISO. Benchmarked on T1 with 8 threads.

3.3.1 CalculiX's Matrix Format

The simulation software CalculiX uses a rather exotic sparse matrix format that excels in storing (structurally) symmetric matrices as they arise from FEM simulations [Dho17]. A structurally symmetric matrix has non-zero entries at positions that are mirrored along the diagonal. In contrast to ordinary symmetric matrices, these mirrored values can be different. Apart from two things, the format used by CalculiX resembles the CSC format. Firstly, all of the diagonal entries are stored in an individual array. Secondly, the upper triangular entries are stored in row-major order after the lower triangular values, which are stored in column major order. The number of values and offsets that have to be stored for symmetric matrices is given by $2 \cdot m + 1 + \frac{nnz}{2}$, when nnz is the number of off-diagonal non-zeros. Structurally symmetric matrices require an additional $\frac{nnz}{2}$ values. CalculiX's representation of the matrix from equation (2.1) is provided in table 3.2.

This storage format conveys advantages and disadvantages:

1. The row indices only have to be stored for values of the lower triangular matrix instead of all values.

2. In case the matrix is not only structurally but fully symmetric, values of the upper triangular matrix can be omitted, further decreasing the memory demands.
3. Diagonal values are efficiently accessible. For FEM applications, the diagonal entries should not be zero and thus, no unnecessary values are stored.
4. The values are stored in a combination of row-major and column-major order. Consequently, they are not suited to be processed by standard implementations of operations like sparse matrix-vector or matrix-matrix multiplication.

Table 3.2: Matrix storage in CalculiX's format.

columnptr	1	3	4	4	4	4	4
row	3	6	4				
diagonal values	1	0	2	3	10	0	
L values	5	8	7				
U values	4	6	9				

3.4 Sparse Direct Solvers

All efficient LU-based solvers use nested dissection as the foundation of their algorithms. It splits the matrix intro partitions which allows supernodal and multifrontal approaches. Using the latter, the update tasks are not performed immediately but only once for every target block. To accomplish this, contribution blocks have to be stored for every update task [DRS16]. This requires more effort and does not necessarily pay off for single node parallelization. Therefore, the focus of this work will be on supernodal solvers, as they require less organizational overhead and the scalability is sufficient for shared memory and accelerator-based implementations.

Solvers that are based on multifrontal methods are, for example, MUMPS [Ame+00] and SPOOLES [AG99]. Both have been around for more than 20 years. In contrast to SPOOLES, MUMPS is still maintained and optimized. PARDISO [SG04] and PaStiX [HRR02] are known for supernodal methods. They have also been in development for over 20 years. PARDISO has been regarded as the fastest solver for indefinite systems in the past [SHG04].

All mentioned solvers are parallelized for shared memory, distributed memory or both models.

3.4.1 Accelerator Implementations

In theory, LU factorization should be portable to GPU architectures with modest effort. The arithmetic intensity is high and required BLAS 3 operations such as `GEMM` and `TRSM` perform very well. There are two problems why there are only few sparse direct solver implementations for GPUs:

1. The memory of the GPU is much smaller than the main memory. No matter how sophisticated the reordering algorithm is, a sparse direct solver will in general produce fill-in. Using state-of-the-art graph partitioning libraries, the fill-in for our asymmetric benchmark matrices is on approximately 15 and tends to increase with the size of the matrix. For the benchmark matrices, this means that the storage of *LU* possibly exceeds 50 GB. Many modern GPUs specialized for HPC provide 12 to 16 GB; a few provide 32 GB.
2. The sparsity lowers the arithmetic intensity. While static blocking strategies for dense solvers always allow ideal block sizes for large BLAS 3 operations, sparse solvers have to work with the blocks provided by nested dissection. The problem will quickly become memory-bound, when mostly small blocks are formed. One could allow more fill-in to obtain a coarser blocking and therefore better performance. This, however, stands in conflict with the first problem.

This implies that for a full acceleration of the program only small matrices can be used as input. This is not feasible because of two reasons: A modern solver should be able to handle all the matrices currently considered state-of-the-art. Moreover, small matrices lead to lower arithmetic intensity and the whole point of GPU accelerations is lost. This is not completely true for the solving step of direct solvers. Existing GPU libraries offer fast triangular solves for applying *LU* on many right-hand sides. Given that there are enough right-hand side vectors, high arithmetic intensity is guaranteed. For the Newton-Raphson method considered in this work, the system has to be solved for only one right-hand side in each iteration. This makes factorization the dominating part in computational complexity.

As part of this work, we tested the GPU-solver SSIDS [Bav16], that provides a GPU implementation of the LDLT algorithm. We did not receive a valid result in reasonable time for the symmetric matrices M1 and M3.

Due to a lack of full-GPU parallelizations for the LU algorithm, we tested the QR implementation of cuSolver. Compared to the LU factorization it requires more operations but therefore provides numerical stability and allows non-squared matrices.

Unfortunately, the solver provided by Nvidia, cuSolver, could not compute our sample matrices due to a lack of GPU memory.

3.4.2 Hybrid Implementations

For the applications considered in this work, a hybrid strategy is currently the most effective way to implement a GPU-accelerated direct solver. It solves the previously identified problems:

- Operations on small blocks that have low arithmetic intensity can be performed on the CPU.
- The data for operations on large blocks can be transferred to and from the GPU when needed. Device memory is freed once associated operations on the GPU finish. The matrices can consequently be as large as the main memory allows.

Nevertheless, the approach also introduces a new problem. As mentioned in section 2.3, the PCI Express bus is the bandwidth bottleneck. Compared to a full-GPU parallelization, the hybrid implementation challenges this bus far more:

1. The dense matrix blocks, which include the fill-in, have to be transferred from the CPU to the GPU. In a GPU-only version, the transformation from sparse to dense could be done on the GPU and only the sparse matrix would be sent via PCI Express.
2. The dense matrix blocks will be sent one-by-one leading to multiple small transfers instead of a single large transfer. The overall throughput will be less due to organizational overhead and latency.

Research on hybrid implementations has been done, but in most cases the development did not go beyond experimental stage [Luc+10; KP09; Geo+11; CW+11]. Two established libraries have introduced optional GPU offloading strategies. Like most experimental hybrid solvers, SuperLU [Li05; SVL14] performs threshold based offloading, in which the scheduling between GPU and CPU is done statically and based on the arithmetic intensity of individual operations. A performance model that considers FLOP/s and memory transfer can theoretically determine, whether GPU or CPU can perform an operation faster. Eventually, this leads to a scheduling in which blocks beyond a certain threshold size are sent to the GPU. The hybrid version of SuperLU did, in contrast to the CPU version, not return valid results. Therefore we did not perform further tests with SuperLU.

The second established LU factorization library that features optional GPU offloading is PaStiX. As this work centers mostly around PaStiX, possible improvements to it and its integration in CalculiX, we introduce and analyze it separately in section 3.6.

3.5 Sparse Iterative Solvers

When implementing iterative solvers there is usually not much room for fundamental innovation. The most popular and versatile algorithms based on Krylov subspace methods, such as GMRES and BiCGSTAB, were introduced in the 1990. Variations of these are a must have for larger iterative solvers. At least as important as the iterative method is the preconditioner. That is where the libraries differ. Most libraries offer diagonal and ILU preconditioning; some offer SPAAI or Multigrid methods.

Adapting the general iterative methods for GPU architectures is trivial because they mostly rely on sparse matrix vector multiplications (`SpMV`). Compared to its older sibling, `GEMM`, this operation has a constant arithmetic intensity. It is still a great application for the GPU because the `SpMV`s work on the same matrix, which merely has to be transferred to the device once. As a result, many GPU-supporting iterative libraries exist.

Before testing the performance difference between these solvers, we have to show that the iterative method shows convergence in reasonable time. Therefore we benchmark GMRES with block Jacobi and ILU preconditioning for M2. The state-of-the-art solver PARDISO can solve this matrix directly in less than 40 seconds. We apply the OpenMP version of the iterative solver Ginkgo [Anz+19] to measure which precision we can achieve iteratively within 40 seconds.

The results visualized in figure 3.2 show very slow convergence behavior that stagnates before reaching acceptable performance. The block Jacobi preconditioning improves the convergence speed slightly. Unfortunately, the ILU preconditioning does not provide any advantage. While ILU(p) methods with higher levels of fill-in could further improve the effectiveness, it also introduces new problems. Instead of one large `SpMV` operation, small triangular solves are performed which leads to slower performance. Moreover, with more allowed fill-in, the memory requirements to the device are growing.

We performed further tests, including the libraries AMGX [Nau+15] and MAGMA [Agu+09] but the results were very similar to the ones presented in figure 3.2. Estimations on the conditioning of our benchmarking matrices returned condition numbers between 10^{10} and 10^{14}, which we consider mediocre but not ill conditioned. We believe

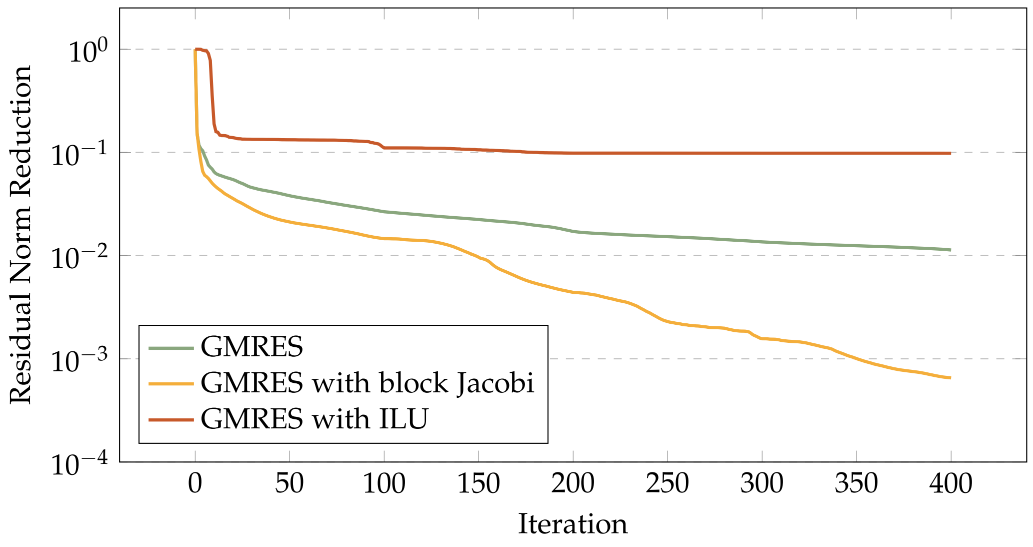

Figure 3.2: Convergence behavior for GMRES with different preconditioning strategies.

that the matrix has to be arranged differently by the simulation software to make iterative solvers more effective. Since this exceeds the scope of this thesis, we will further focus exclusively on direct solvers.

3.6 PaStiX

PaStiX is a direct sparse linear equation solver that was developed by researchers at INRIA around 2000 [HRR02; Pic+17]. The algorithms provided by the library are Cholesky, Hermitian, LDLT and LU factorization. It was improved constantly over the past 20 years and the result is a highly optimized solver. PaStiX uses a supernodal right-looking approach. The procedure is conventional for a direct solver:

1. Determine a reordering permutation to reduce the fill-in and generate a potential block data structure.
2. Analyze the problem through symbolic factorization.
3. Factorize numerically by applying Cholesky, Hermitian, LDLT or LU factorization.
4. Solve the system through forward and backward substitution.
5. Refine iteratively in case the solution is numerically inaccurate.

During the first decade of its development, PaStiX was parallelized for shared and distributed memory with POSIX threads and MPI. Later, the library was extended by task based parallelization through PaRSEC and StarPU [Lac+14; Lac15]. This conveys two major advantages:

- The functionality and hardware-aware optimizations are only loosely coupled, which allows easier maintainability with regards to performance portability. The PaStiX researchers showed that their task-based approach with PaRSEC, did not introduce significant overhead compared to the previous NUMA-aware pthreads implementation.
- By splitting the work and data into many small tasks, a selection of those can be scheduled to the GPU. In PaStiX these tasks consist of `GEMM`s and `TRSM`s.

Even though offloading compute-intensive tasks to the GPU is not a new idea [Luc+10; KP09; Geo+11; CW+11], it has not be done before in a task-based fashion for a supernodal direct solver. First, we assess how PaStiX performs for the benchmark matrices. To achieve proper results we use a sophisticated guess for tunable parameters. The specific parameter selection and the reasoning behind it are discussed in section 5.1. The performance results for M1 to M9 are shown in figure 3.3. PaStiX clearly outperforms PARDISO with CPU and hybrid mode for cases in which many operations are required, such as for M3, M5, M6 and M9. For other matrices, the libraries perform similarly. In those cases, the GPU provides less to no speedup. Since only the factorization step is GPU-accelerated, we measure it separately to get more insight in the offloading behavior.

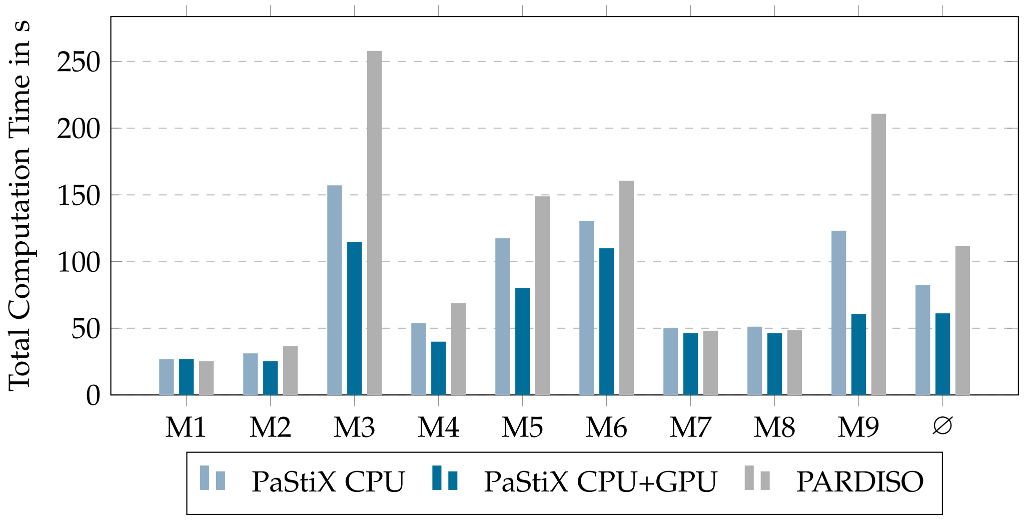

Figure 3.3: Baseline comparison of PaStiX and PARDISO.

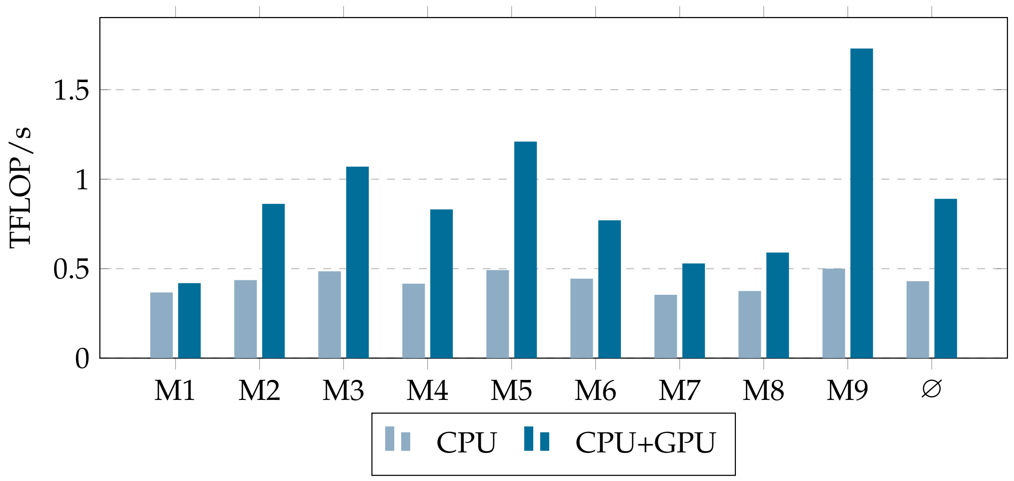

Figure 3.4: Baseline comparison of PaStiX's CPU and hybrid performance during factorization.

The results visualized in figure 3.4 show that the CPU performs around 500 TFLOP/s and the hybrid version up to 1.8 TFLOP/s. While the CPU reaches more than 50% of its theoretical peak performance, the hybrid version is far off from the potential 7.8 TFLOP/s of the V100. To explain this gap we will construct an approximate performance model to calculate an upper bound with regards to the PCI-Express bottleneck. Therefore, recall that

$$FLOP_{\texttt{GEMM}}(n) = 2n^3 \qquad FLOP_{\texttt{TRSM}}(n) = n^3$$

For this performance model we must not use the computational intensity because that considers all memory transfers and we only want to consider those between the CPU and GPU. PCI Express provides independent bidirectional transfer. Thus, we only have to consider the initial or final transfer. Moreover, we assume that multiple tasks associated with the same supernode are performed on the GPU. For M9, we know that the ratio of `TRSM` to `GEMM` operations is approximately 1 to 3. This is exactly the situation visualized in figure 3.5. To execute all those tasks on the GPU, 49 blocks of size n^2 have to be transferred. The number of FLOPs is $12 \cdot FLOP_{\texttt{TRSM}}(n) + 36 \cdot FLOP_{\texttt{GEMM}}(n) = 84n^3$. The computational intensity with regards to the PCI Express transfers is consequently

$$\frac{84n^3}{8 \cdot 49 \cdot n^2} = \frac{3n}{14} \text{ FLOPs per byte.}$$

Given the block size n and the bandwidth between CPU and GPU β, an upper bound for the FLOP/s the hybrid version can perform when only restricted by the PCI Express bus is

$$P_{Hybrid} \approx P_{CPU} + \frac{3n}{14} \cdot \beta.$$

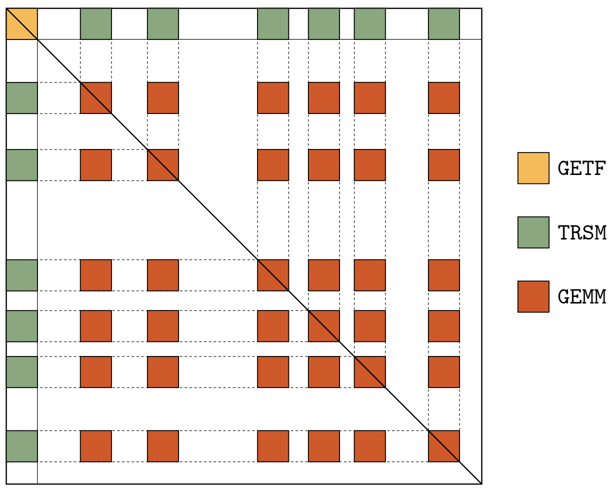

Figure 3.5: Tasks associated to one supernode. The LAPACK operation `GETRF` is always executed on the CPU. Only `TRSM` and `GEMM` can be offloaded.

To identify reasonable values for the unknowns in the above equation, we have to make further approximations.

1. As shown in figure 3.4, $P_{CPU} \approx 500$ GFLOP/s $= 500 \cdot 10^9$ FLOP/s.
2. The maximum block size n is 1024. This limit is based on the parametrization discussed in section 5.1. A higher limit would decrease the number of tasks and therefore the degree of parallelism.
3. The unoptimized bandwidth for host to device memory transfer is approximately 7.0 GB/s. This is based on the measurements performed in section 5.2. Thus, $\beta \approx 7.0 \cdot 10^9$.

Using these values, the approximated upper bound for performance in FLOP/s is

$$P_{Hybrid} \approx 500 \cdot 10^9 + \frac{3 \cdot 1024}{14} \cdot 7.0 \cdot 10^9 = 2.036 \cdot 10^{12} \text{ FLOP/s} \approx 2 \text{ TFLOP/s}$$

The performance model reaches higher FLOP/s than any of the benchmarks. With 1.73 TFLOP/s M9 is the closest. The other matrices are far off. However, recall that the performance model is very approximate and only an upper bound. It assumes that large blocks are ready to be processed constantly and the performance is only restricted by the PCI-E bandwidth. In reality many of the blocks are smaller than 1024, which reduces the computational complexity. Small blocks result from geometries that are easy to partition. Large blocks originate from strongly connected clusters in the geometry. Especially leafs and lower nodes of the elimination tree tend to be small and large blocks are only ready for computation once their children in the elimination tree are completed [Lac+14].

The performance decisive parameters β and n cannot be optimized trivially. The blocks are generated based on the reordering. Enlarging them artificially does increase the performance but not necessarily the total computation time since more unnecessary instructions are performed. The ratio between `TRSM` and `GEMM` operations also impacts the performance but it depends on the sparsity pattern and cannot be altered either. Merely the bandwidth β can be optimized. This will be discussed in section 5.2. Furthermore, we can double the computational intensity by computing the factorization in single instead of double precision. This idea is pursued in section 4.1.

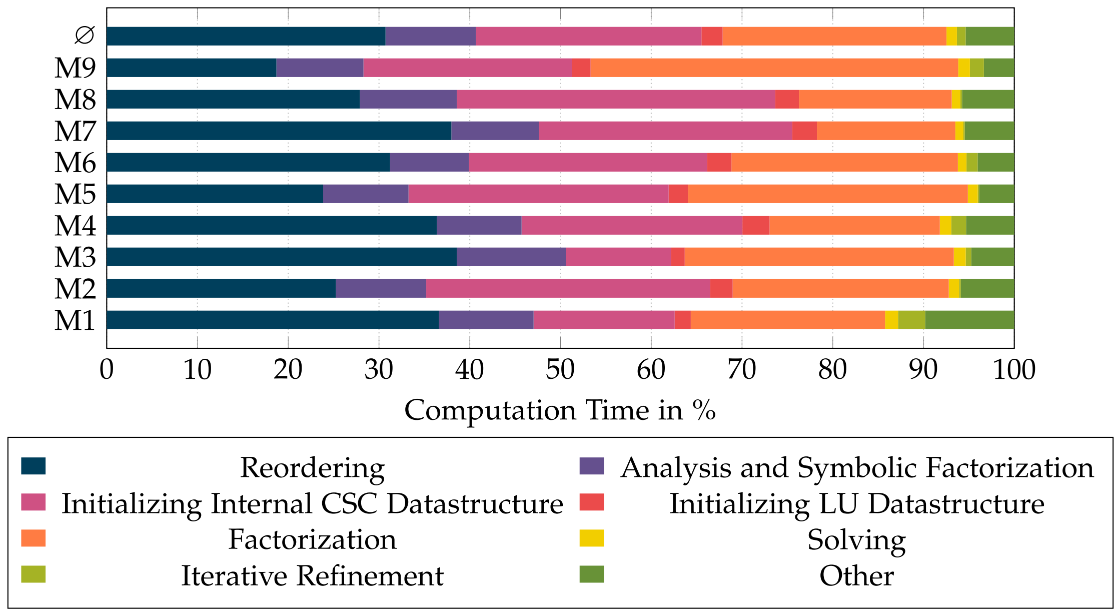

Figure 3.6: Structure of PaStiX's computation time in hybrid mode.

Besides the already very optimized factorization, PaStiX has to perform many other steps. Figure 3.6 shows how much time is spent for the benchmark matrices in each step. In its current state, PaStiX spends only a quarter of the total computation time in the factorization step. To use PaStiX efficiently we have to optimize and parallelize the remaining three quarters. Section 5.3 elaborates on this process for the initialization of the internal CSC data structure. Other steps that are difficult to optimize, such as the reordering, analysis and symbolic factorization, are dealt with in section 6.2.

4 Mixed Precision in PaStiX

Mixed precision methods switch between different floating point data types during the execution of an algorithm. Common precision levels are half, single and double. The motivation is simple: In most processing units two single precision operations can be executed instead of one double precision operation. For GPUs the same relation extends to half and single precision.

For direct solvers, a common strategy is to perform the factorization in lower and the iterative refinement in higher precision [But+07; Bab+09; Hai+18]. Thus, the factorization runs faster and by performing the refinement in higher precision, the final result should be as accurate as in fixed precision computation with high precision. The key factor for achieving speedup through this technique is fast convergence during iterative refinement. Otherwise it might require more time than is gained by the faster factorization.

In this chapter we explore the possibilities of implementing a mixed precision strategy in PaStiX. Section 4.1 introduces and evaluates a hybrid implementation for single and double precision. Section 4.2 discusses the option to perform selected operations on the GPU in half precision.

4.1 Single and Double Precision

PaStiX offers computation in either single and double precision, but not mixed precision. As shown in figure 4.1, the factorization performs much faster in single than in double precision.

Simulation tools in structural mechanics usually run in double precision because the Newton-Raphson method might converge slower and high accuracy is desired in general anyways. Using the pure single precision mode of PaStiX for such applications would have two disadvantages:

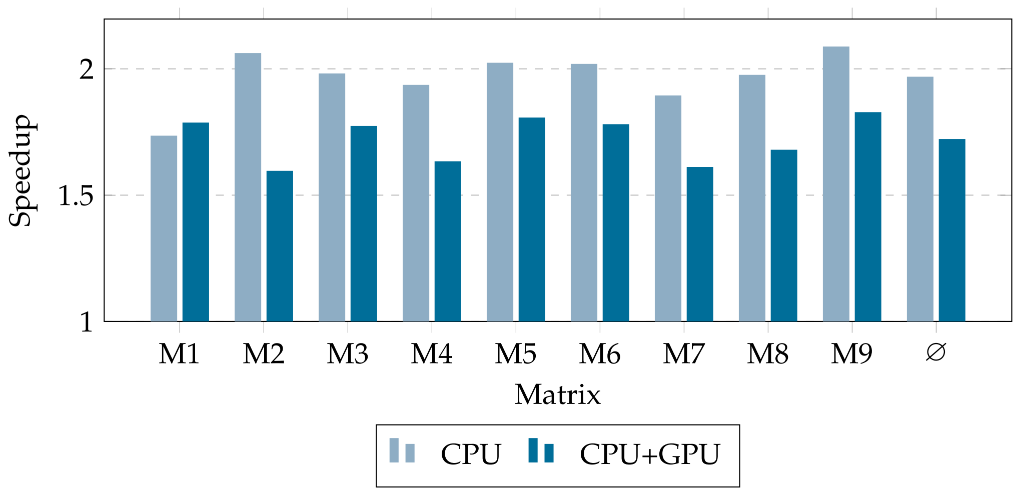

Figure 4.1: Speedup by performing the factorization in single instead of double precision.

1. The iterative refinement converges slower, because the arising numerical errors are much larger.
2. The result obtained from iterative refinement might reach the targeted residual of 10^{-12}. This does not imply that a residual of 10^{-12} is achieved with regards to the matrix in double precision.

A mixed precision approach for PaStiX is to perform the factorization and the solving in single precision, followed by the iterative refinement in double precision. PaStiX implements the right preconditioned GMRES algorithm presented in section 2.5.3. The preconditioner $M = LU$ is very effective because it is, apart from numerical errors, identical to A. The challenge for a mixed precision implementation is to incorporate the preconditioner, whose matrices L and U are in single precision, into the refinement in double precision. There are two possibilities:

1. Cast LU from `float` to `double` so that the iterative refinement including the preconditioning can be performed in double precision.
2. Keep the preconditioner in single precision and cast the input vector from `double` to `float`. In each iteration the preconditioner is applied in form of a forward and backward substitution to solve $LUx = b$. Therefore, we can cast x and b to `float` in order to perform the preconditioning in single and the rest of GMRES in double precision.

Both approaches are viable mixed precision methods, but the latter approach surpasses the former in multiple aspects:

- The LU, which consists of dense blocks, requires up to 30 GB for the benchmark matrices in single precision. Casting that many values takes considerable time and possibly negates the advantage gained by mixed precision. Casting a vector of at most 40 MB in each iteration is less time consuming.
- The LU matrix is dominating the total memory consumption and when we perform the factorization in single instead of double precision, the required memory for LU is halved. When the matrix is cast from `float` to `double` afterwards, the memory advantage is lost. Instead, the required memory capacity is now one and a half times more, because during the casting LU is kept in both accuracies.
- Applying the preconditioner in double precision might reduce the number of required iterations. Nevertheless, this effect is very weak because the LU still originates from single precision factorization. On the contrary, the speedup achieved by executing the forward and backward substitution in single precision is almost 2. Therefore, it outweighs the disadvantage of slightly more iterations.

We implement the mixed precision approach in which the input vector is cast from `double` to `float` and back in every iteration. As shown in table 4.1, the number of required iterations to reach a residual of 10^{-12} is much higher with the new mixed precision approach. However, this is to be expected since both the initial solution vector and the preconditioning are less accurate. GMRES converges for each of our benchmark matrices in less than 20 iterations.

The results presented in figure 4.2 show that the new mixed precision implementation is performing better than the original solver with fixed double precision. However, while the CPU version achieves a speedup between 13.27% and 84.57%, the hybrid version performs only between 6.83% and 49.24% faster than its fixed precision equivalent. Firstly, it can be observed that mixed precision performs better for more time consuming factorizations, such as for M3, M5, M6 and M9. Moreover, the matrices for which many iterations of refinement are required, such as for M1, M7 and M9, obviously demand

Table 4.1: Required iterations in double and mixed precision.

Matrix	1	2	3	4	5	6	7	8	9
Double Precision	1	0	0	1	0	1	0	0	0
Mixed Precision	8	6	6	5	4	4	8	6	19

more time for solving and refinement. Since the backward and forward substitution that is applied for solving and preconditioning is not accelerated by the GPU, the hybrid version performs especially poorly in these cases. The lack of a hybrid implementation is discussed in section 5.4. Finally, the GPU also achieves less speedup because the factorization itself does not benefit from the switch to single precision as much as the CPU, which shown in figure 4.1.

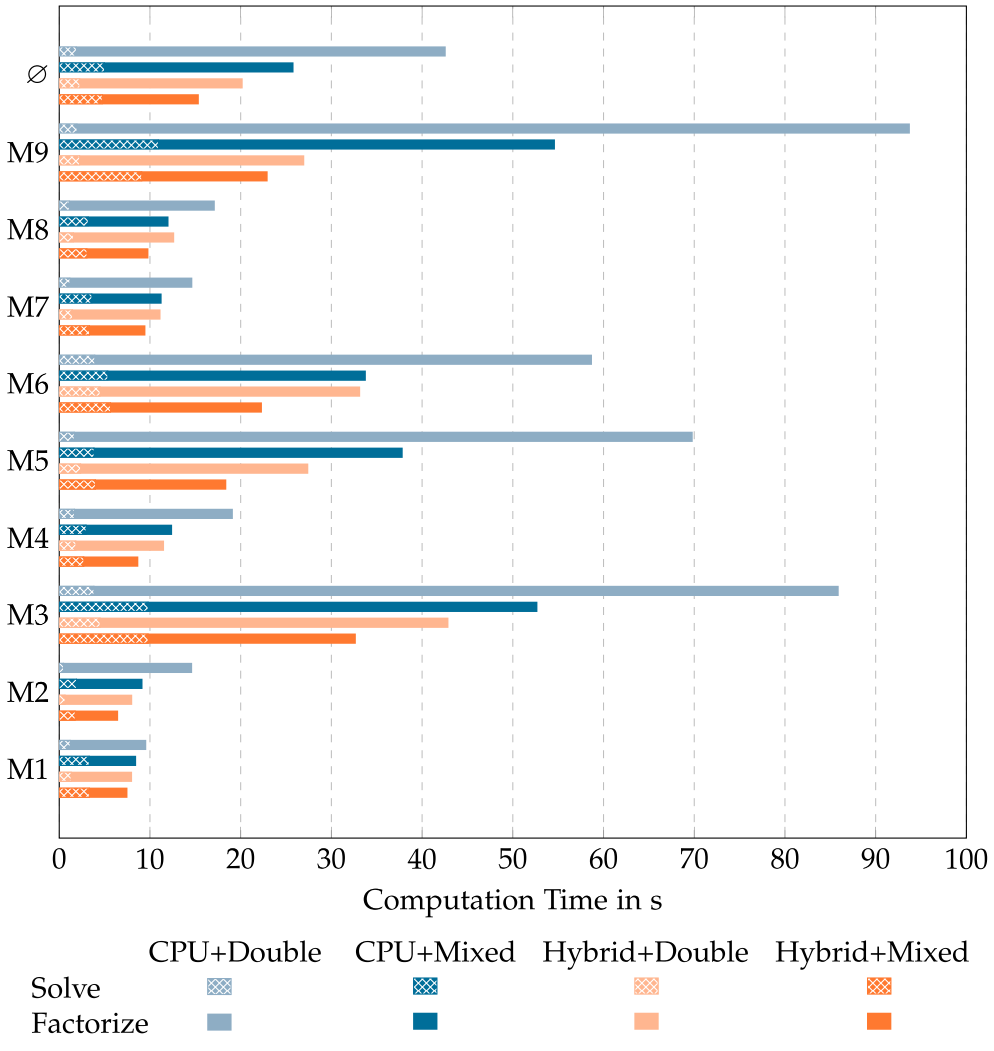

Figure 4.2: Impact of the mixed precision implementation on CPU and hybrid solver. *Solve* times include solve and iterative refinement step. *Factorize* times include internal CSC initialization, *LU* initialization and factorization step, as all of those are influenced by the new precision scheme. The benchmarks are performed on T1 with 8 threads and optionally with Nvidia V100.

4.2 Half Precision

After successfully introducing a mixed precision approach in which the factorization is done in single and the refinement in double precision, a next step is to further lower the accuracy of the factorization to half precision (FP16). Half precision is not supported by ordinary CPUs but with the rising popularity of machine learning, it became a popular feature in GPUs. The NVIDIA V100 of our benchmark system, for instance, can theoretically perform up to 125 TFLOP/s in half precision [NVI17]. In case of PaStiX, the computationally dominant operation during the factorization is `GEMM`, for which an efficient half precision implementation exists in cuBLAS (`hgemm`).

The 125 TFLOP/s are performed partially by so-called tensor cores (TC), that only work with 16-bit inputs. They can perform `GEMM`s in which the scalar multiplications are calculated in FP16, but the accumulation in FP32. Therefore, it produces more accurate results than the pure FP16 operation. This approach has been tested before on MAGMA, a library that solves $Ax = b$ directly for dense matrices, and achieved a speedup of up to 4 [Hai+18]. The expectations for PaStiX in the context of structural mechanics simulations are significantly lower for two reasons:

1. PaStiX is a solver for sparse matrices and has to deal with lower computational intensity compared to MAGMA.
2. In [Hai+18], only matrices A with condition numbers $\kappa(A) \leq 10^6$ were considered. The matrices we are benchmarking as part of this work have mediocre conditioning ($\kappa(A) > 10^8$) and we expect them to converge slower during iterative refinement. This is underlined by [CH17], in which convergence for GMRES in half precision is only guaranteed for ($\kappa(A) \leq 10^8$).

Furthermore, we agree with the decision made in [Hai+18]. Only `GEMM`s should be performed in half precision, as this is usually the dominating computational part and its conditioning is better than `TRSM`'s. The program's structure with respect to precision is therefore:

$$\text{Factorization}\begin{cases}\textbf{(FP32)}\ \texttt{GETRF}\text{ for small diagonal blocks}\\ \textbf{(FP32)}\ \texttt{TRSM}\text{ for contributions}\\ \textbf{(FP16)}\ \texttt{GEMM}\text{ for updates}\end{cases}$$

(FP32) solve by applying LU
(FP64) GMRES with **(FP32)** preconditioning

To perform `GEMM` in half precision, casting it from single precision is required. We do so on the GPU after the blocks in single precision have been transferred because it is easier to implement. A more efficient approach concerning the memory bandwidth between CPU and GPU would compress them before the transfer to GPU in order to relieve the PCI Express bus. The casting is performed with the command `__half2float` and `__float2half` provided by CUDA. To test the numerical behavior for half precision computations we test one version which uses $\texttt{FP16} \cdot \texttt{FP16} = \texttt{FP16}$ and a second one that uses $\texttt{FP16} \cdot \texttt{FP16} = \texttt{FP32}$ provided by the tensor cores. These operations are titled `cublasHgemm` and `cublasGemmEx` in the cuBLAS library. For solving the benchmark matrices we scale them by a factor of 10^{-5} because otherwise the numerical values would exceed the range of FP16, which is $-65504 \leq x \leq 65504$. As shown in table 4.2, the tensor core multiplication introduces far less additional iterations than the pure half precision operation.

Table 4.2: Iterations until a residual of 10^{-12} is reached with the two half precision operations in comparison to the FP64/FP32 implementation.

Matrix	1	2	3	4	5	6	7	8	9
`cublasSgemm`	8	6	6	5	4	4	8	6	19
`cublasGemmEx`	400+	6	6	6	6	6	10	8	25
`cublasHgemm`	400+	36	75	54	57	60	400+	400+	400+

Up to this point, the half precision implementation is a proof of concept to analyze the numerical behavior. The version is not optimized for efficiency and performance measurements are not meaningful. For the following reasons we decide to not pursue the idea of half precision further:

- Even though the tensor core matrix multiplication showed promise with regards to the numerical behavior, fast convergence could not be guaranteed for every test case.
- The scaling would require a sophisticated approach. A numerical analysis has to be performed in advance, based on which a customized scaling or normalization technique could be performed.
- The current bottleneck of the application is the PCI Express bus, not the computational power. In our proof of concept, the matrix is still transferred to the GPU in single precision state. No matter how fast the casting and tensor core multiplication is performed by the GPU, the PCI Express bottleneck still exists. Therefore, an approach to this problem would be to cast the values to half precision before sending them to the GPU. A problem is that normally `GEMM`s are executed subsequently to `TRSM`s so that the data already resides on the GPU. Conducting the

initial transfer in half precision would require that `TRSM` is performed in single precision as well. Moreover, using the tensor core operation the result has to be transferred back in single precision anyways so that the actual bottleneck still exists for the return transfer. Considering this, only true half precision can relief the PCI Express bus.

- With our implementation enabling the tensor cores to unlock the maximum potential of the Nvidia V100 is not possible. The deployment of tensor cores in cuBLAS is limited to `GEMM`s where `K`, `LDA`, `LDB` and `LDC` are multiples of 8 and `M` is a multiple of four. To fulfill these criteria, padding has to be inserted into PaStiX's dense block data structure.

The implementation of the above mentioned features and optimizations could potentially lead to an overall performance gain for matrices that do not demand much iterative refinement. Nevertheless, the required effort exceeds capacity of this work.

5 Optimizations for PaStiX

In this chapter we discuss several independent ideas for optimizations in PaStiX that increase the performance in different ways. First, section 5.1 elaborates on parameters exposed by PaStiX and how to tune them for applications in structural mechanics. In section 5.2, we discuss the utilization of CUDA's pinned memory to improve the performance during factorization. Section 5.3 covers the OpenMP parallelization of the sparse matrix preprocessing. Finally, Section 5.4 introduces a partially accelerated iterative refinement implementation.

5.1 Parameter Tuning

PaStiX offers a variety of parameters that impact the overall performance. Many are negligible or related to unused features. Most important for our purposes are the following five parameters:

1. Selection of a partitioning library (SCOTCH [PR96] or METIS [KK98]).
2. Selection of a scheduling library (PaRSEC [Bos+13] or StarPU [Aug+11]).
3. Upper bound for the width of supernodes, called column blocks in the context of PaStiX' data structure (`MAX_BLOCKSIZE`).
4. Splitting width for column blocks (`SPLIT_SIZE`).
5. Column width threshold for enabling 2D instead of 1D tasking (`TASKING_THRESHOLD`).

The first two parameters are easy to choose because they are binary choices. Whether SCOTCH or METIS performs better for our benchmarking matrices is mainly impacted by two factors. The time required for computing the reordering and number of operations that have to be performed during factorization. For M1 to M9, these characteristics are shown in figure 5.1. While the reordering permutations produced by the libraries lead to a similar factorizations, SCOTCH performs the nested dissection based algorithm much faster than METIS. Therefore, we decide to use it for further purposes.

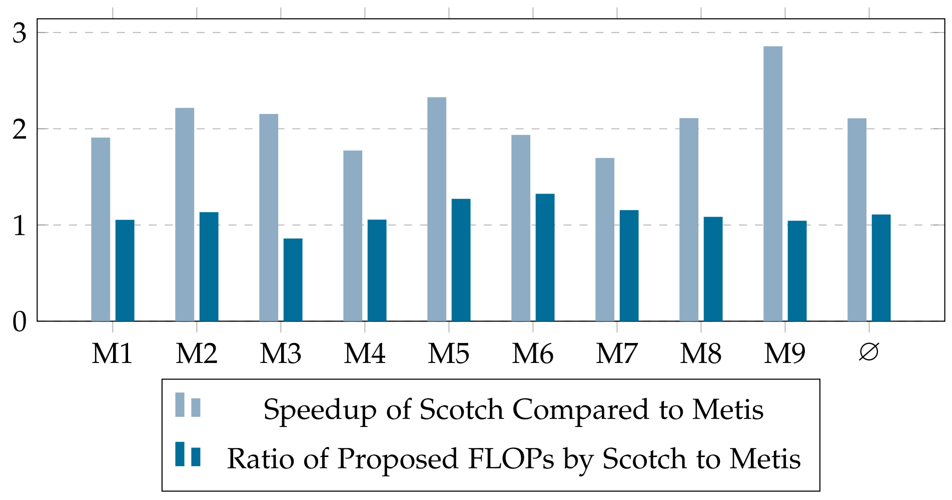

Figure 5.1: Comparison of SCOTCH and METIS for the benchmarking matrices. SCOTCH is on average twice as fast but produces slightly worse reorderings.

The second binary choice is between the two scheduling libraries PaRSEC and StarPU. We set this parameter based on the performance during factorization. As shown in figure 5.2, PaRSEC clearly outperforms StarPU in hybrid execution mode. When restricted to CPU computation, PaRSEC is only slightly faster.

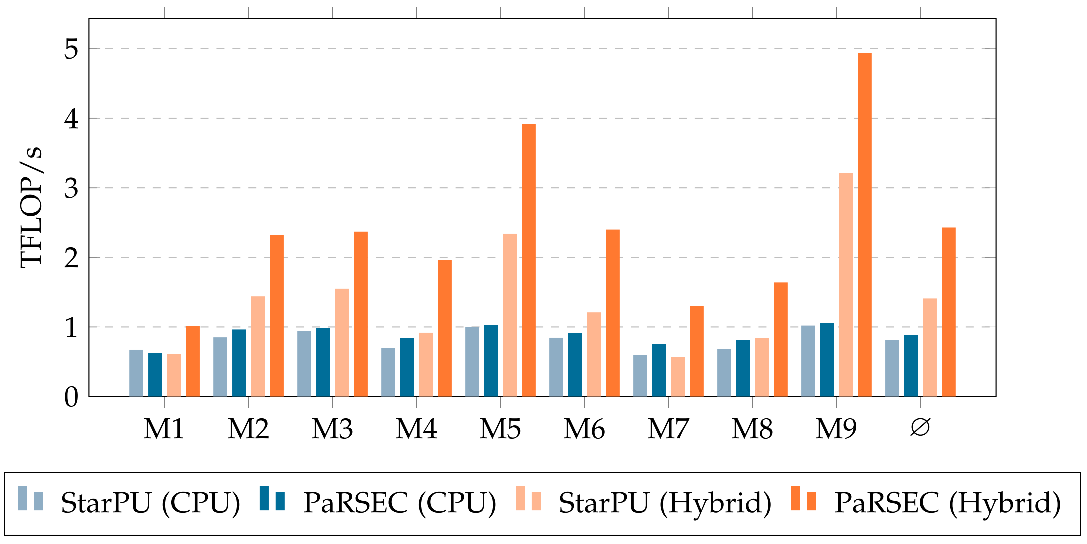

Figure 5.2: Performance comparison of StarPU and PaRSEC in single precision.

The remaining three parameters are integer values and depend on each other. The PaStiX data structure for *LU* consist of column blocks that are formed based on the elimination tree generated by nested dissection. The initial structure may look like the left matrix of figure 5.3. The problem with this blocking is that the amount of parallelism is limited since only 3 tasks exist. Furthermore, the load cannot be distributed evenly because the load for each task varies heavily. To avoid such behavior, large column blocks are further divided into smaller column blocks. The rule for this splitting is that every block having a width larger than `MAX_BLOCKSIZE` is split into as many chunks so that each has a width larger or equal to `SPLIT_SIZE`. To increase the amount of parallelism even more, the 1D tasks are split into 2D tasks. To avoid the creation of many small tasks, which would lead to significant scheduling overhead, the 2D splitting is performed only on column blocks with more than `TASKING_THRESHOLD` horizontal entries.

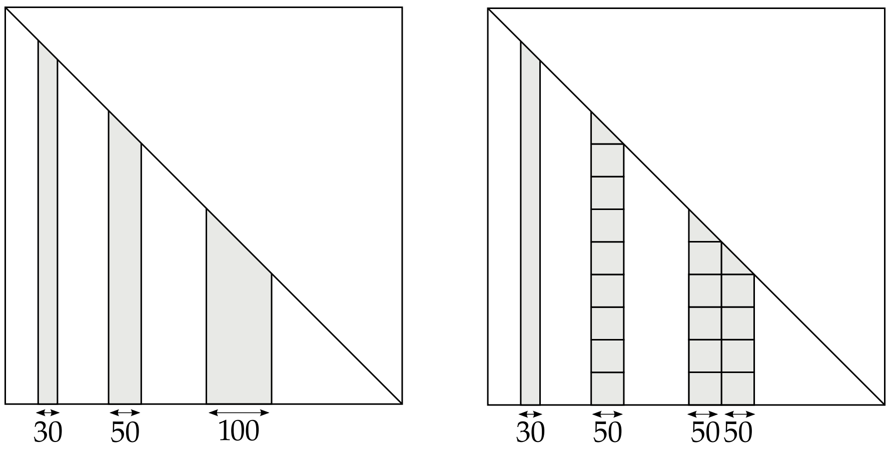

Figure 5.3: Effects of splitting parameters with `MAX_BLOCKSIZE` = 60, `SPLIT_SIZE` = 50 and `TASKING_THRESHOLD` = 40. For simplicity, we assume that the matrix is dense. The first block is smaller than `MAX_BLOCKSIZE` and `TASKING_THRESHOLD`. Therefore, it is not split. The second block is split into multiple 2D tasks as it is larger than `TASKING_THRESHOLD`. The rightmost block is further split into two column block of size `SPLIT_SIZE` because it is larger than `MAX_BLOCKSIZE`.

With the standard configuration of PaStiX (`MAX_BLOCKSIZE`=320, `SPLIT_SIZE` = 160, `TASKING_THRESHOLD`=160) our benchmarks perform poorly, especially when enabling offloading to GPU. This is because the default parameter configuration leads to many small blocks, which is great for load balancing but devastating for the arithmetic intensity of BLAS 3 operations. We found much better configurations (`MAX_BLOCKSIZE`=2048, SPLIT_SIZE = 1024, TASKING_THRESHOLD=128) that massively increased the FLOP/s for the factorization. With such a configuration, 2D splittings are still performed for smaller columns but larger columns are not split as much anymore.

A model that predicts the performance of the factorization based on a certain splitting is desirable but not feasible to implement because it varies from matrix to matrix. The approach of manual experiments that led to the configuration mentioned above is not very satisfying either. To assess the problem scientifically, we perform automatic tuning for the three parameters. For this, we use the external python module OpenTuner [Ans+14], which offers a wide variety of machine learning-based optimization algorithms for the purpose of automatic tuning. We only have to provide the range of the three input parameters and specify how the feedback value is returned to the library. In this case, the feedback value is the computation time required for the factorization.

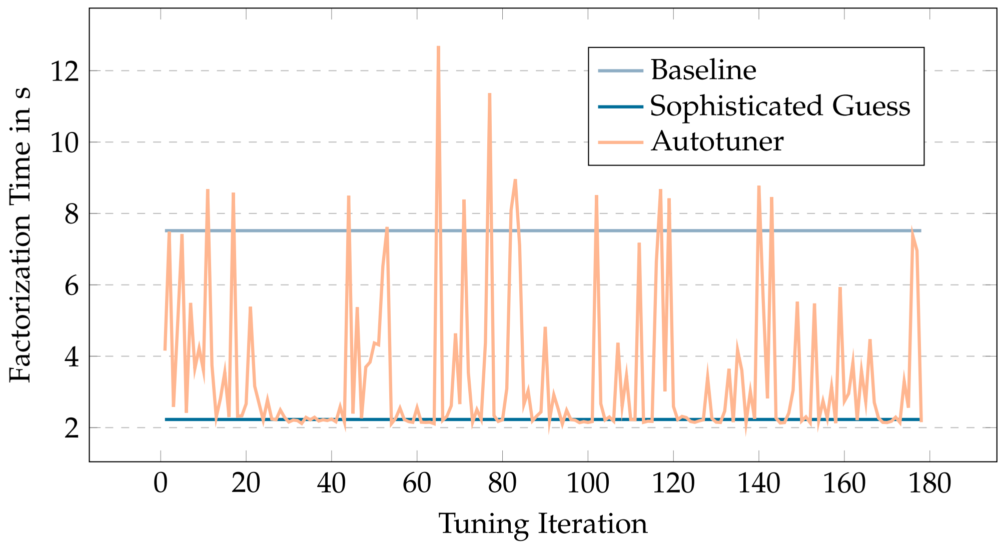

Figure 5.4: Progression of automatic tuning with OpenTuner using M2 as input. Benchmarked on T1 with 8 threads and GPU support.

The results of the automatic tuning shown in figure 5.4 indicate that our sophisticated guess is close to the optimal configuration. Only a few runs of the auto tuner were slightly faster. Because the optimal blocking parameters also depend on the specific problem, a more sophisticated parameter tuning is desirable for future application.

5.2 Pinned Memory

As introduced in section 2.3.1, host and device address spaces are disjoint. The CPU cannot access device memory and the GPU cannot access host memory. The exchange of data is performed via `cudaMemCpy`. The default process for host to device transfer consists of two steps. First, the CPU allocates a separate buffer and fills it with data to be sent. Only then the GPU is allowed to copy data from the buffer to GPU memory. The intermediate buffer is required because otherwise the data might be evicted from main memory while it is transferred to the GPU. It is non-pageable memory and therefore called *pinned* memory [Wil13].

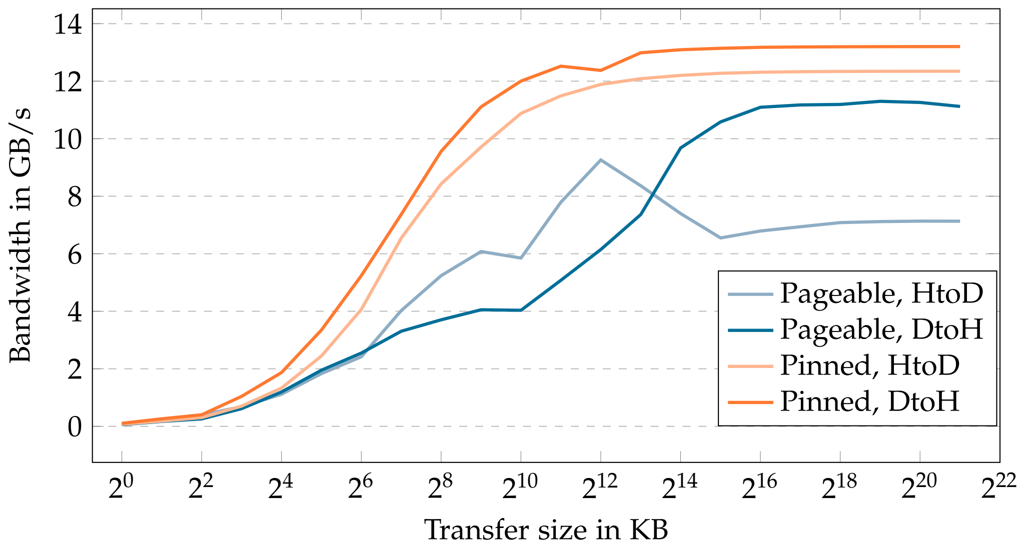

Figure 5.5: Memory bandwith with and without pinned memory. Device to host transfers achieve higher bandwidth than the host to device transfers.

It is possible to allocate pinned memory manually, so that additional buffers are not required and the GPU can copy the data via direct memory access (DMA). Therefore, the memory has to be allocated by the host with `cudaMallocHost` instead of a regular `malloc`. As shown in figure 5.5, the bandwidth for large chunks almost doubles for host to device communication and slightly increases for device to host communication when using pinned memory. The main drawback is that the allocation is costly. As shown in figure 5.6, allocating 6 GB of pinned memory takes approximately one second. 6 GB/s is slightly slower than memory transfer with ordinary pageable memory and therefore, allocating memory with `cudaMallocHost` is only advantageous when data is transferred from or to a pinned buffer multiple times.

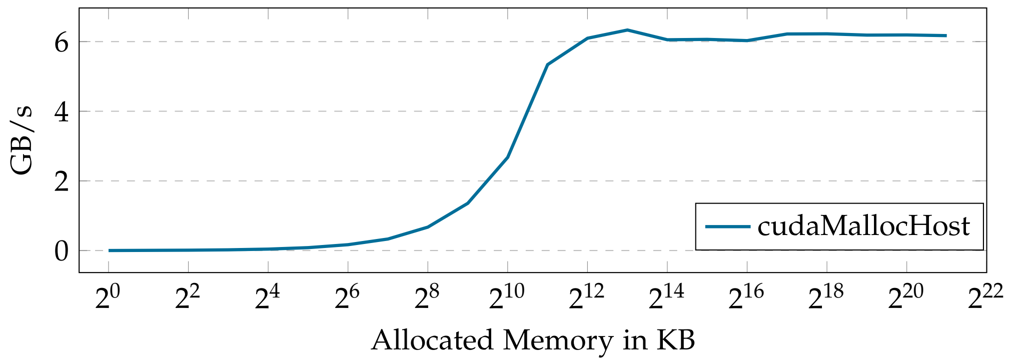

Figure 5.6: Performance of allocating pinned memory using `cudaMallocHost`.

PaStiX does currently not use pinned memory optimizations, even though the factorization benefits from higher CPU-GPU bandwidth, as shown in figure 5.7. The tasks that are already scheduled to the GPU would be pushed to and popped from the device faster. Moreover, smaller tasks which have previously been computed on the CPU, might be scheduled to the GPU now because the penalty for memory transfer is less significant. The problem, however, is that data from the pinned buffer is not transferred to the GPU more than once. Even worse, since not all of the blocks are computed on the GPU, some elements in the allocated memory range are never send to the GPU. This means that pinned memory does not provide any overall speedup, since allocating takes more time than is gained by faster transfers.

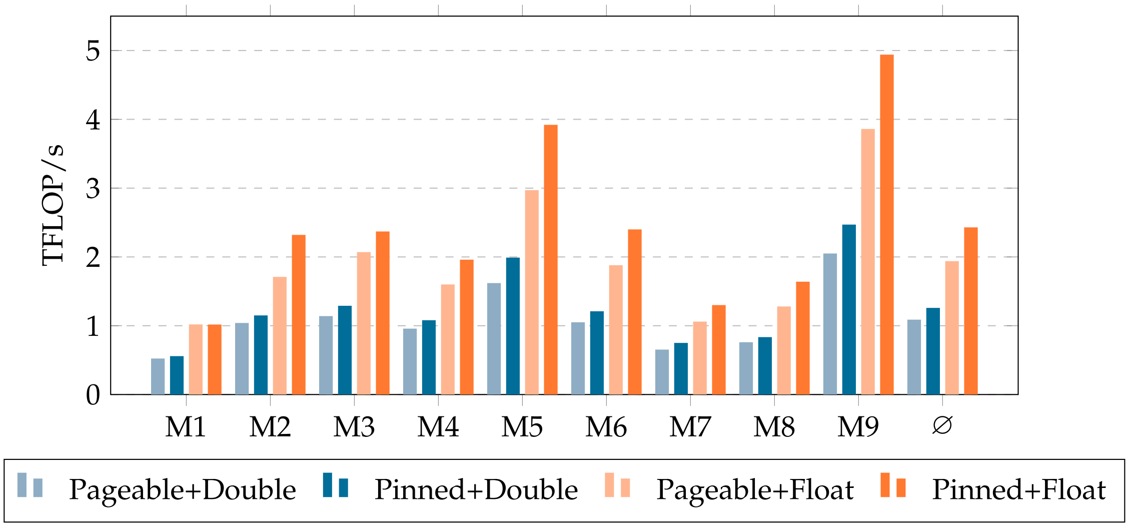

Figure 5.7: Performance impact of pinned memory on PaStiX's factorization.

The drawback of allocating pinned memory ranges can be amortized when the same buffer is used for more than one factorization process. Iterative methods, such as the Newton-Raphson method, are used in simulation tools to approximate non-linear equations. In each iteration a linear equation has to be solved and therefore, the expensive to allocate pinned memory can be reused. Since the required buffer size might vary slightly between the iterations, an overestimation of the first iteration's demands should be used to avoid situations in which the memory buffer has to be reallocated. For the benchmark inputs of this work, the overhead of pinned memory allocation and faster factorization are presented in table 5.1. With these values a break-even-point that shows how many iterations are required for pinned memory to be faster in total computation time, can be computed. This break-even-point is reached after at most 7 iterations for double precision and even earlier for single precision.

Table 5.1: Computation of a break-even-point for the usage of pinned memory with matrices in double precision.

Matrix	1	2	3	4	5	6	7	8	9
Storage in GB	8.0	14.8	41.6	18.8	51.6	59.1	19.1	21.5	47.5
Time for Allocation in s	1.3	2.4	6.8	3.1	8.4	9.7	3.3	3.5	7.8
Factorization Advantage in s	0.29	0.48	3.59	0.86	3.44	2.67	0.74	0.54	3.57
Break-Even-Point	4.48	5	1.89	3.6	2.44	3.63	4.46	6.48	2.18

5.3 Parallel Matrix Preprocessing

In PaStiX there is an important step prior to the factorization. It is titled "Initialize Internal CSC" and as shown in section 3.6, it requires between 11% and 35% of the computation time, which in many cases is longer than the actual factorization. The tasks performed in this step are:

1. Structuring the matrix in a block-scheme based on the analysis.
2. Applying the reordering permutation to the sparse matrix.
3. Transposing the sparse matrix in order to process the upper triangular matrix U analogously to the lower triangular matrix L and to use it during iterative refinement for row-major based algorithms.

The first task is negligible with regards to performance and therefore only the latter two tasks are discussed in this section. The general flow of these tasks is not optimal in the original PaStiX implementation, so that the reordering was applied to both the source and the transposed matrix. Moreover, the mixed precision approach introduces a new challenge to this step. The reordered and transposed matrix has to be available in single precision for factorization and in double precision for iterative refinement. That is why we propose a new task flow that minimizes the use of expensive permutation and transpose functions. As shown in figure 5.8, reordering and transposing only have to be performed once. Merely the casting which is computationally low-cost has to be performed twice.

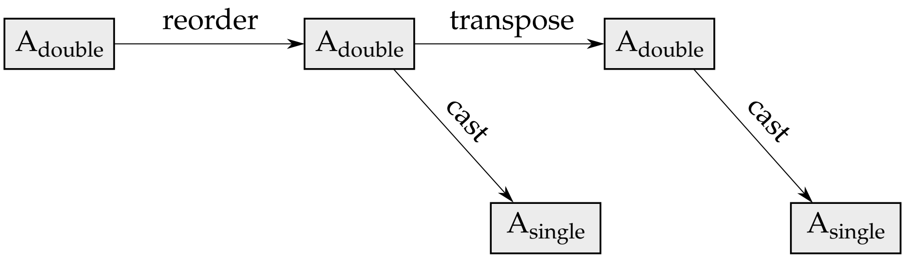

Figure 5.8: Task Flow for Matrix Preprocessing

5.3.1 Parallel Permutation of a Matrix in CSC Format

The application of the reordering itself is a symmetric permutation of the sparse input matrix. Given a permutation vector p, this affects every array of the CSC format introduced in section 2.2:

- The changes in `colptr` are trivial. The number of elements in each row can be computed based on `colptr`. These can then be permuted and accumulated to form a new `colptr`.
- The changes to `rows` are trivial only on first sight. Each row index can be mapped accordingly and no copies have to be created. However, initially the entries per column in `rows` were sorted in ascending order. After the mapping, this is not the case anymore. PaStiX and many other libraries demand that elements are stored in order so that the matrix can be iterated over meaningfully. Given that each column has an entry, the number of vectors that have to be sorted equals the number of equations.

- The `values` always have to match the indices of `rows`. Since `rows` was rearranged, `values` has to be permuted identically.

To optimize the application of the reordering, we parallelize the sorting invocations using OpenMP. Additionally, we replace the default C sort function `qsort` with C++'s `std::sort`. The sort invocation is supposed to permute `values` analogously to `rows`. This can be achieved by sorting an array of ascending indices with regards to `rows` and applying the resulting permutation to the both arrays. The scaling of the implementation is evaluated in section 5.3.3.

5.3.2 Transpose of a Structurally Symmetric Matrix

Computing the transpose of a structurally symmetric matrix in CSC format is the same task as converting a structurally symmetric matrix from CSC format to CSR format. The CSC's `colptr` is the CSR's `rowptr` and analogously the CSC's `rows` are CSR's `cols`. What has to be done computationally is restructuring the `values` from column-major to row-major order. This problem is very fundamental and has been analyzed thoroughly for asymmetric matrices [Wan+16]. The fact that the pattern is symmetric makes the problem much easier and most high-level libraries targeting sparse matrices do not offer the standalone functionality. Moreover, we do not want to introduce further external dependencies for such a minor task. Therefore, we present our own approach to this problem.

To swap each entry with their diagonally mirrored entry, the values have to be iterated. This happens in a nested loop construction displayed in listing 5.1. The outer loop iterates over the number of columns n. Based on `colptr`, the inner loop iterates over the non-zero entries of each column. The known facts for the currently iterated entry are the column of the entry, the number of preceeding entrys in the same column and the row of the entry through `rows`. With this information, we have to acquire the same data for the mirrored entry. The column is easy to deduct because the row of the source element is the column of the target element. The difficult part is figuring out the offset to the first entry of the row. We know the column of the source element and therefore the row of the target element, but not how many non-zero entries in the same column precede it. In listing 5.1 we show two approaches to solve that problem. One is fast but strictly sequential, while the other one has a worse computational complexity but allows parallelism.

The first algorithm makes use of the fact that the loop iterates the matrix sequentially columnwise from the leftmost column to the rightmost column. Using that information,

```c
memset(temp, 0, sizeof(int) * colptrIn[n]);
for(int i = 0; i < n; i++){
    for(int j = colptr[ i ]; j < colptr[ i + 1 ]; j++){
        // Strictly Sequential Version
        offset[j] = temp[ rows[ j ] ]++;

        // Parallelizable Version
        int length = colptr[ rows[ j ] + 1 ] - colptr[ rows[ j ] ];
        int* rowsWithOffset = rows - colptr[ rows[ j ] ];
        offset[j] = binarySearch( rowsWithOffset , 0, length - 1, i);
    }
}
```

Listing 5.1: Two approaches to compute the number of preceeding elements in the same row of an entry in a CSC matrix.

the offset to the first entry in each row can be computed. Due to symmetry, this is identical to the targeted offset to the first entry in each column.

The second algorithm utilizes that the target entry's row is known. Based on that, binary search can find the correct position inside the target column. As shown in figure 5.9, the parallel algorithm scales well but even with the maximum number of supported threads, the parallelized algorithm takes more time than the sequential counterpart. On a machine with 16 cores the implementations would most likely perform equally. For now, we choose the sequential implementation.

Once the target position for each element is known, the transpose is a simple mapping that can be parallelized trivially using OpenMP.

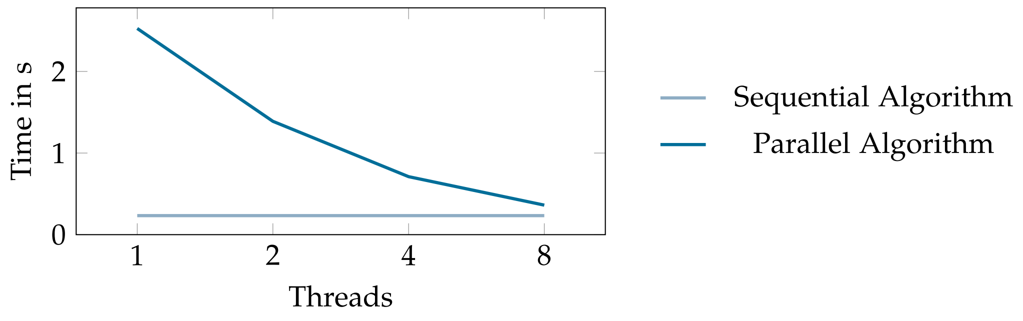

Figure 5.9: Performance comparison of the two approaches for computing row offsets. The measurements are based on M2 executed on T1.

5.3.3 Performance Evaluation

Besides the algorithms introduced in the previous two sections, casting from `double` to `float` is also performed in parallel. The graphs in figure 5.10 show that reordering and casting scale well, considering that this is obviously a memory-bound problem. Since we decided to use the sequential algorithm for computing the transpose, only the swapping of values is parallelized. Thus, the poor speedup when executing with 8 threads was expected. The parallel efficiency of the total step is 50% when using all of the available cores.

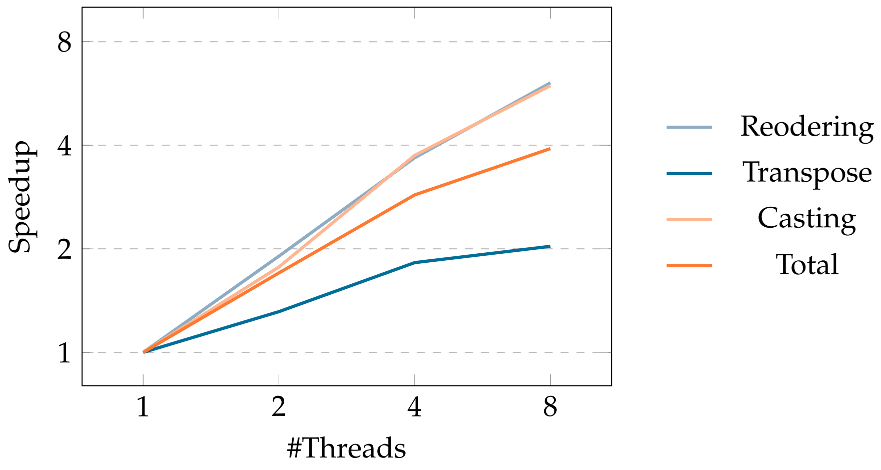

Figure 5.10: Speedup for shared memory parallelization of maxtrix preprocessing. The measurements are based on M2 executed on T1.

Reusing Capability

In chapter 6, a concept will be introduced in which PaStiX is called repeatedly with the same sparsity pattern but different numerical values. In such cases, we have to perform the reordering and the computation of the transpose only partially because some of the circumstances have not changed. The application of the reordering was dominated by sorting row indices. The sorting delivered a permutation vector based on which `rows` and `values` are reordered. Once the vector has been computed, it is correct as long as the sparsity pattern of the input matrix does not change. By keeping this permutation array between PaStiX invocations in memory, we avoid sorting and therefore lower the computational complexity of the step. A similar approach can be used for computing the transpose which was dominated by finding the indices of mirrored elements. The

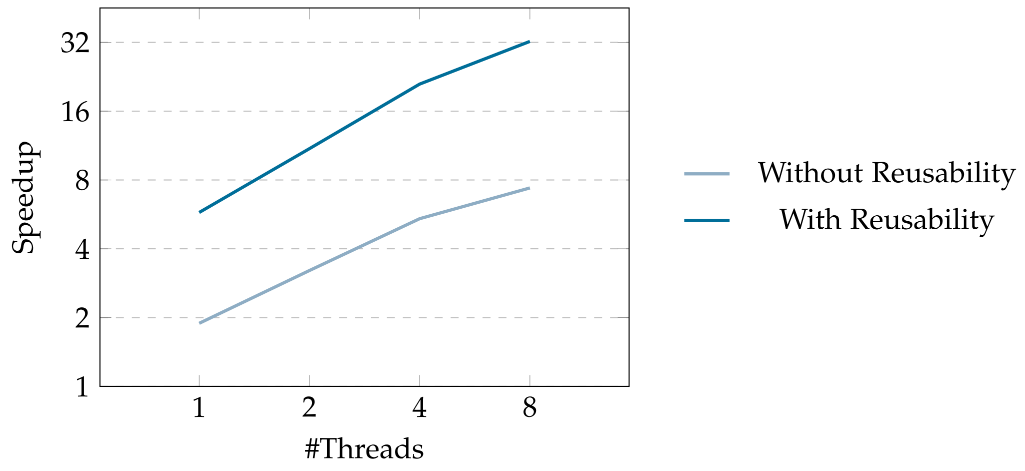

Figure 5.11: Performance comparison of the matrix preprocessing. The baseline for the calculated speedup is the original (sequential) PaStiX code. *With Reusability* assumes that the permutation order for the reordering and the column offsets for the transpose are known. The measurements are based on J2 executed on T1.

offset to the first entry in each row can be kept in memory so that it does not have to be computed again in further PaStiX invocations with the same sparsity pattern.

The data visualized in figure 5.11 shows that the entire step is now up to 7.3 time faster compared in the original implementation. When PaStiX is called subsequently with the same sparsity pattern, a speedup between 5.7 with one thread and 32.3 with 8 threads over the original implementation is achieved.

5.4 GPU-Accelerated Iterative Refinement

The successful introduction of a mixed-precision strategy has the side effect that more iterations of iterative refinement are required. As visualized in figure 4.2, the time spent on iterative refinement is only slightly less than the time spent on the factorization. Until now, it is merely parallelized with POSIX threads for the CPU. A GPU implementation does not exist. In theory, iterative methods for linear systems of equations do benefit from accelerators, as discussed in section 2.5. A problem is the need of a precisely approximating preconditioner. In section 3.5 it was determined that a very effective preconditioner is required for the benchmark matrices to converge reasonably fast. The ILU preconditioning tested as part of this work was not a precise approximation and did not improve the convergence behavior. PaStiX's preconditioning strategy is to use the factorized matrices L and U as preconditioners. Because of numerical errors during the factorization, this preconditioner is not totally accurate. At most one iteration is required for the benchmark matrices, when double precision is used for the factorization and less than 20 iterations when mixed precision is used.

As introduced in section 2.5.2, the preconditioner has to be applied in each iteration in form of forward and backward substitutions, further named *solving*. GMRES without preconditioning is dominated by sparse matrix vector multiplication (`SpMV`) whose computational complexity is constant. The same does apply for the triangular solves applied to the preconditioner. The pitfall is that for our benchmark matrices the LU used by the preconditioner has 7 to 20 times more entries than the sparse matrix A that is used for `SpMV`. Therefore, the computation time of the iterative refinement step is clearly dominated by the application of the preconditioner.

An efficient GPU implementation of the iterative refinement would perform as many operations as possible on the device. Unfortunately, the solving that is part of preconditioning cannot be reasonably accelerated. As shown in table 5.1, LU is very large and generally does not fit into memory, especially when considering that the sparse input matrix has to be stored in addition. Moreover, the constant computational intensity makes it a very inconvenient application. Consider M6 for which LU requires 25.8 GB of memory in single precision. Given a bandwidth of 12.35 GB/s, the transfer time is 2.1 seconds. The CPU requires in total 3.7 seconds for the iterative refinement. This means that the GPU implementation must be more than twice as fast as the CPU implementation, in order to achieve an overall speedup.

Therefore, we accelerate every operation of GMRES except those concerning the preconditioning. The existing iterative refinement implementation already offers a pseudo-object oriented interface that can be extended by GPU functionality. As introduced in listing 2.1, the operations required by the algorithm are `SpMV`, `DOT`, `AXPY`, `SCAL`, `NRM2`, `GeMV`, `ROT`, `ROTG` and `TRSV`. Except `SpMV`, those functions are provided by cuBLAS and are integrated in a straight-forward approach. For the `SpMV` operation we deploy LightSpMV [LS15]. At the time of its release, the authors claimed that it performs better than CUSP and cuSPARSE [LS15]. In contrast to PaStiX, LightSpMV requires the sparse matrix in CSR format. The conversion is analogous to the implementation in section 5.3.2. The library excels with its dynamic scheduling that balances the load much better than the static scheduler used in the original implementation of `SpMV` in PaStiX. In general the right-preconditioned GMRES algorithm, which was introduced in section 2.5.3, is computationally structured as follows:

1. Compute r_0 and v_0 using `SpMV`, `norm2`, `scal`.
2. Apply preconditioner using `solve` and subsequent `SpMV`.
3. Perform Arnoldi Process using `dot`, `axpy`, `norm` and `scal`.
4. Find y_m by applying Givens rotations using `rot` and `rotg`.
5. Obtain the final x_m using `trsv` and `gemv`.

As mentioned earlier, the main problem is that the `solve` operation ($w = M^{-1}v$) cannot be performed efficiently on the GPU. Therefore, the vector w is transferred from GPU to CPU. Then, the CPU-based `solve` algorithm is applied and subsequently the result v is sent back to GPU. Considering that `solve` operates on multiple GB of data, the transfer times of the two vectors is negligible.

A general prerequisite for executing the sparse matrix vector multiplication on the GPU is that the input data (A, x and b) are in the device's memory. For x and b the transfer times are once more negligible but the matrix A in CSR format and double precision requires noticeable time. The data sizes and estimated transfer times are shown in table 5.2. One key strategy when programming for GPUs is to hide memory transfers by concurrent computation. In this case, the regular solving has to be performed prior to the iterative refinement in order to obtain an initial solution. We use `cudaMemcpyAsync` to asynchronously transfer the sparse matrix to the GPU while performing `solve` on the CPU. A synchronizing barrier is placed before the first appearance of `SpMV` for which the sparse matrix is required as input.

Table 5.2: GPU transfer times for sparse matrices in comparison to the `solve` times.

Matrix	1	2	3	4	5	6	7	8	9
Memory in GB	2.14	1.93	6.55	2.30	5.68	6.99	3.16	4.20	3.32
Estimated Transfer Time	0.30	0.27	0.92	0.32	0.80	0.98	0.44	0.59	0.47
`solve` Time	0.34	0.34	0.96	0.38	0.57	0.71	0.37	0.38	0.64

Performance Evalutation

First, we measure the new implementation's performance under the assumption that no preconditioning has to be applied. This is the classical GMRES procedure in which we expect the GPU to heavily outperform the CPU. The data visualized in figure 5.12 supports this hypothesis. The GPU implementation achieves speedup between 9.3 and

16.9 compared to the the CPU version. Without preconditioning the algorithm is not interesting for practical purposes. Therefore, the hybrid algorithm with preconditioning and initial device memory transfers is evaluated next. The data in figure 5.13 shows that the hybrid implementation is faster than the pure CPU version. The advantage, however, is not very significant because the preconditioner, which accounts for a large fraction of the total time, is still performed on the CPU. Nevertheless, the new implementation is faster for all benchmark matrices and we include it in the productive implementation.

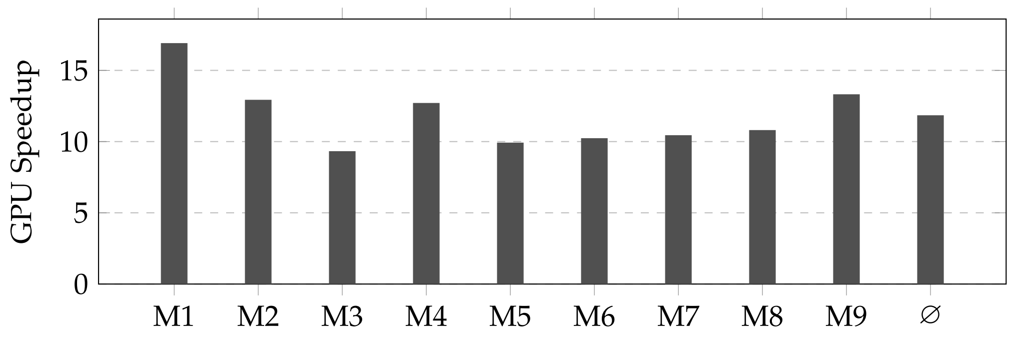

Figure 5.12: Performance comparison of CPU and GPU implementations of GMRES in PaStiX without preconditioning. Executed on T1 with 8 threads.

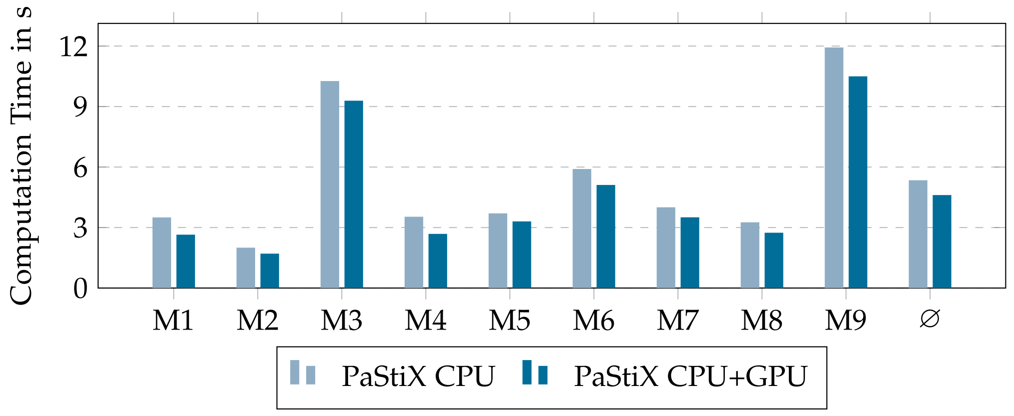

Figure 5.13: Speedup for hybrid implementation of iterative refinement with preconditioning. To include memory transfer times, the entire solve- and refinement-step is considered. Executed on T1 with 8 threads.

6 PaStiX Integration in CalculiX

The previous two chapters discuss optimizations for the hybrid solver PaStiX. In this chapter we present its integration into the FEA software *CalculiX*. The loose coupling between CalculiX and external equation solvers makes the integration uncomplicated. Merely the interface between CalculiX and PARDISO has to adapted slightly for PaStiX. A conversion from CalculiX's own matrix format to CSC is necessary. This is discussed in section 6.1. Section 6.2 introduces a method with which data from previous iterations of the Newton-Raphson method can be reused for further computation which leads to significant reduction of total computation time. Finally we evaluate the amortized performance of PaStiX over multiple iterations in section 6.3.

6.1 Conversion of CalculiX's Matrix Format

CalculiX's internal matrix format is very similar to the CSC format which PaStiX uses. They are introduced in section 2.2.3 and section 3.3.1. A conversion to the CSR format, which PARDISO requires, is already implemented but behaves very poorly with regards to performance. Otherwise, we could have applied it with slight modifications because a structurally symmetric CSC matrix is identical to the transpose of a CSR matrix.

In CalculiX's matrix format, the lower triangular values are stored in column-major order and the upper triangular values in row-major order. This means that the upper triangular matrix has to be transposed and then combined with the lower triangular matrix. Since finding the transpose was already discussed and evaluated in section 5.3.2, we can simply apply the same algorithm. Therefore, the performance behavior of the conversion is similar to the algorithm presented in section 5.3.2 and we omit further evaluations.

6.2 Reusing Matrix Patterns

The most significant weakness of PaStiX is that the majority of the time is not spent on the very highly optimized factorization. Considering the optimizations discussed in the previous chapters, the time-wise dominating part of PaStiX is the computation of a reordering performed by SCOTCH with around 40%. Unfortunately, most reordering libraries offer only distributed memory parallelizations. Shared memory or accelerator-based approaches exist but they are mostly considered experimental.

Nevertheless, this performance bottleneck can be mitigated by exploiting a property of solving non-linear equations in CalcuilX. The linear solver is invoked repeatedly to approximate the solution for a system of non-linear equation as presented in section 2.1.3. During these iterations, the pattern of the matrix changes only slightly because the relations between elements remain mostly the same. Merely between contact elements, previously active connections can become inactive and the other way around. For the matrix representation this means that the number of equations remains the same during an invocation of the Newton-Raphson method. The number of matrix entries may increase or decrease. The numerical values always change because of the iterative algorithm. In many cases, entries that were non-zero originally become zero in a subsequent iteration. Then, we can simply invoke PaStiX with the same matrix pattern as before. Values of the matrix entries that became zero have to be set to 0.0 explicitly. Since the sparsity pattern forwarded to PaStiX and SCOTCH is the same, the computed reordering is also the same and we can simply reuse the reordering computed during the previous iteration. The same applies to the analysis and symbolic factorization step of PaStiX.

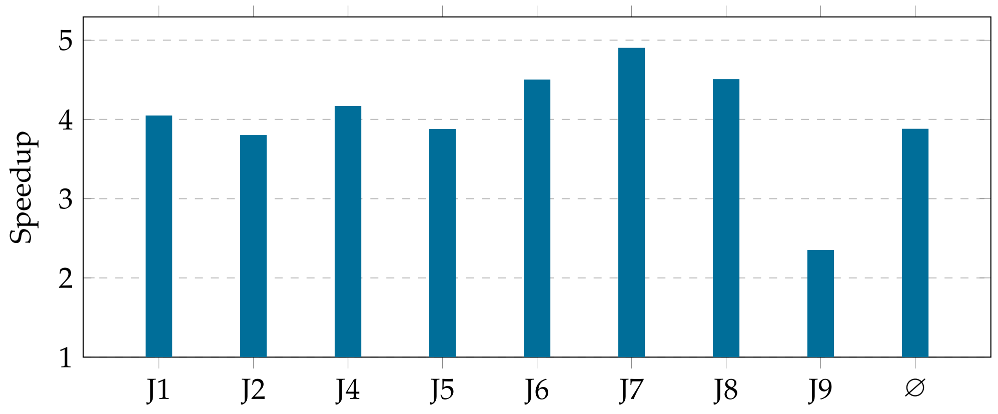

Figure 6.1: Speedup when reusing sparsity patterns compared to when it is not reused.

Naturally, the number of required operations for the factorization might be slightly higher because we consider matrix entries that are zero anyways, but the time saved by skipping the reordering is much more significant, as is shown in figure 6.1. This performance behavior is directly related to the structure of PaStiX's computation time presented in figure 3.6. The input deck J9, for instance, benefits less from this optimization because the factorization step already dominated the computation time in the original version.

To convey the speedup of figure 6.1 to the real application, a high reusability of iterations is required. Reusing the ordering and analysis steps requires that the non-zero positions of the current matrix are a subset of those in the previous invocation. Otherwise the reordering might lead to disastrous fill-in and factorization performance. Moreover, the data structures that are prepared during the analysis step belong to a certain sparsity pattern. The number of iterations in which reusing can be applied with are shown in table 6.1 for the benchmark matrices.

Table 6.1: Reusing capability for the benchmark cases.

Job	1	2	3	4	5	6	7	8	9
Total iterations	7	36	1	125	97	10	165	248	21
Reused Iterations	6	33	0	111	77	5	135	208	15
Reusability	86%	92%	0%	89%	79%	50%	82%	84%	71%

6.2.1 Reusiability Optimization

To further increase the reusability ratio, we have to elaborate on the reason for unstable matrix patterns. An execution of CalculiX usually consists of multiple steps that are specified in an input deck. The steps perform different tasks such as applying individual loads and performing frequency or thermal analysis instead of static analysis. When switching from one step to another, the sparsity pattern and even the number of equations can change and therefore the ability to reuse cannot be guaranteed. Within the steps, a possibly non-linear equation has to be solved approximately and this requires usually around 10 iterations of the Newton-Raphson method. The positioning of non-zero entries changes between iterations because the simulation includes contact areas which were introduced in section 2.1.3. Elements that have been tagged by the user as contact elements are likely to switch between active and inactive during the Newton-Raphson method. To make sure that the sparsity pattern does not change during one step, we assume that every possible contact element is active. This has two side effects:

1. The matrix that is forwarded to PaStiX has slightly more entries than absolutely necessary, which means that more operations have to be performed during the factorization.
2. The algorithm of CalculiX that activates and deactivates contact elements between iterations is not required anymore, which saves time.

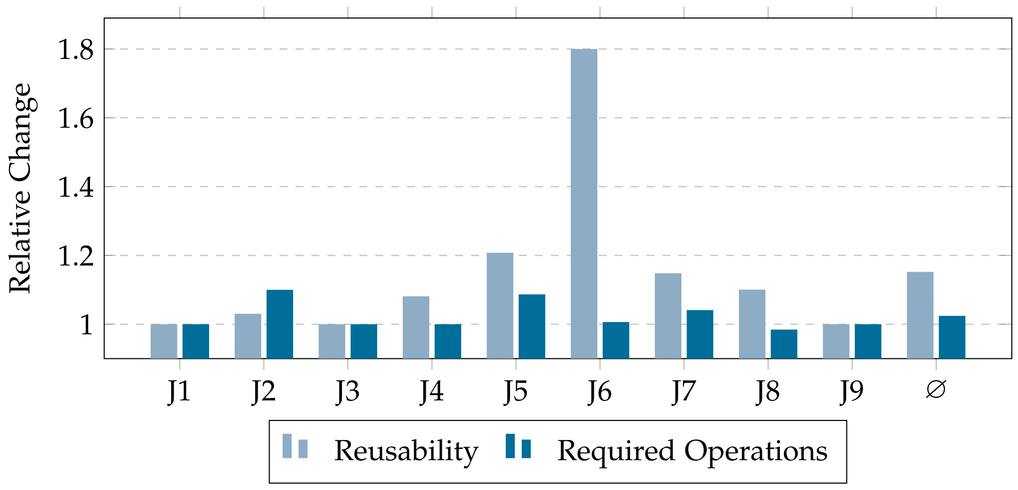

Figure 6.2: Impact of contact approximation on reusability and factorization.

Figure 6.2 shows the gain in reusability compared to the additional number of operations. J1 and J3 do not have contacts areas and thus perform unchanged. The increase in operations is below 10% for every job and therefore negligible, especially when considering that more operations usually lead to more FLOP/s. The number of additional iterations in which reusing can be applied varies from case to case. The highest relative change can be observed for J6 in which 9 instead of 5 out of 10 iterations can be reused.

6.3 Evaluation

In this section, we evaluate PaStiX's and CalculiX's performance with focus on the optimizations performed as part of this work.

6.3.1 Amortized Performance of PaStiX

For analyzing PaStiX in the context of CalculiX and structural mechanics, we focus on the computation time spent on solving the systems of linear equations and ignore the overhead generated by CalculiX. The presented performance measurements in this subsection are therefore calculated by accumulating isolated computation times of PaStiX invocations for each input deck.

One problem identified during the analysis was that numerical computation in form of factorization, solving, and refinement on average merely amounted to 26.8% of PaStiX's computation time. This was less than the time spent on computing a reordering permutation and only slightly more than the time required for initializing the internal CSC data structure. To counter that behavior, we heavily optimized the latter step by enhancing the overall program flow and parallelizing it with OpenMP. We could not optimize the reordering itself but reusing sparsity patterns helped reducing the number of SCOTCH invocations.

Figure 6.3 clearly underlines the positive effects of these implementations. On average, the numerical parts of PaStiX now amount to 66.1% of its total computation time. Only for J3, where reusing sparsity patterns is not possible, the ratio is 25%. The figure also shows the impact of the mixed precision feature which leads to a massive increase of required time for iterative refinement compared to the original measurements visualized in figure 3.6.

Furthermore, we evaluate the performance in comparison to the previously used PARDISO solver. PARDISO's computation times are also amortized. The comparison is not entirely equitable because PARDISO could also benefit from reusing reordering permutations. Nevertheless, the results presented in figure 6.4 do show that the new implementation outperforms the original PARDISO configuration even more than in the initial analysis. Instead of speedups between 0.95 and 3.5, which the unmodified PaStiX library achieved for individual invocations, the optimized hybrid version reaches amortized speedups between 2.6 and 12.7. The CPU implementation is 44% faster on average when using mixed precision features. When offloading to the GPU, this effect reduces to 23%. The advantage of GPU offloading is visualized in figure 6.5. The average speedup is 1.7 in mixed and 2.1 in double precision. The symmetric jobs J1 and J3 achieve the least speedup. J3 only consists of one iteration and is therefore not eligible for pinned memory and reusing features. The small J1 suffers from low computational intensity.

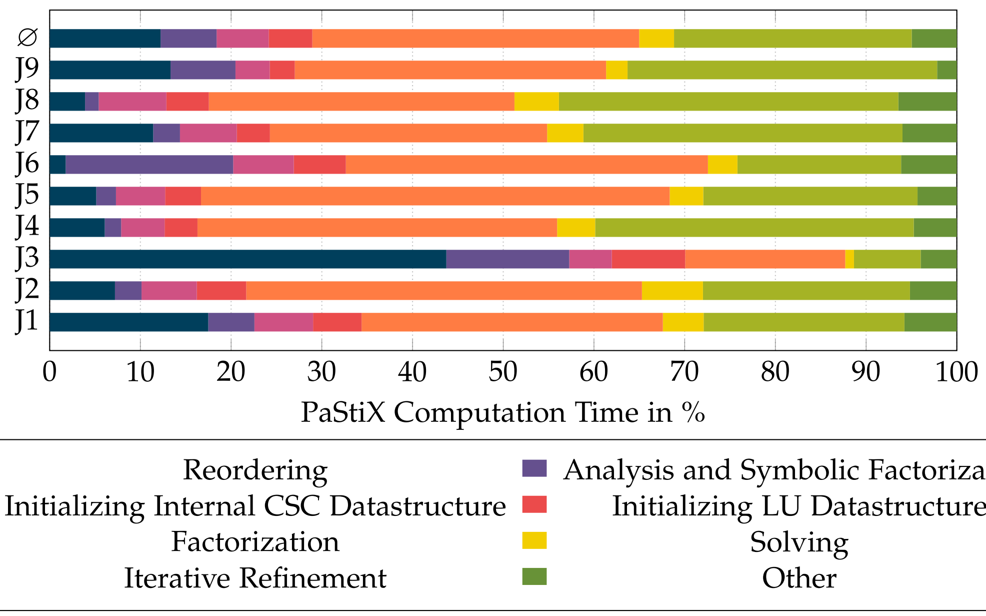

Figure 6.3: Amortized structure of PaStiX's computation time in hybrid and mixed precision mode. Executed on T1 with 8 threads.

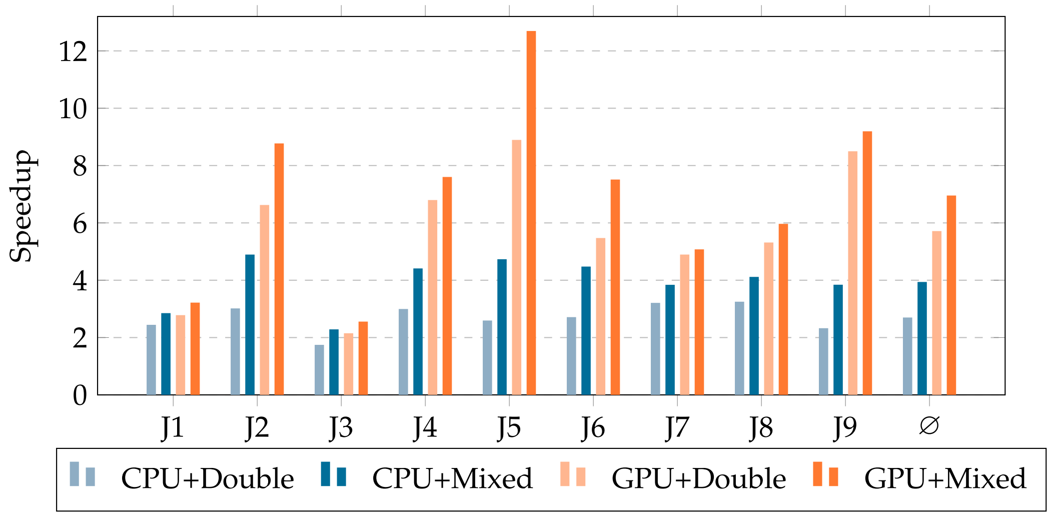

Figure 6.4: PaStiX's performance for CPU and hybrid version in mixed and double precision. The baseline is PARDISO's amortized runtime. Executed on T1 with 8 threads.

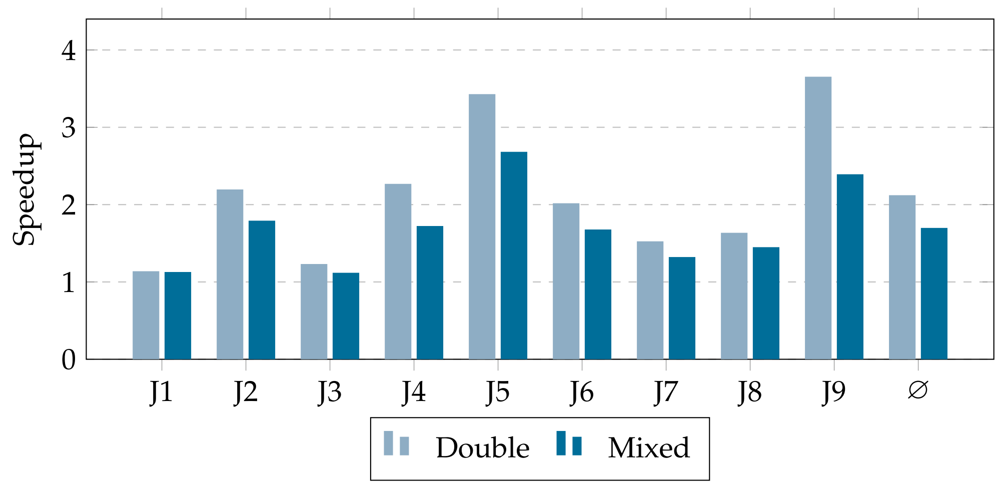

Figure 6.5: GPU speedup of PaStiX's amortized performance in CalculiX. The speedup is computed based on the identical implementation just without GPU offloading. The computation times are amortized over as many iterations as required for each input deck.

6.3.2 Total CalculiX Acceleration

To assess the impact of the new solver we measure the performance of CalculiX with PaStiX integrated. For the original implementation, PARDISO's computation time on average amounts to 71.8% of the total time, which is visualized in figure 3.1. The updated ratios for the hybrid PaStiX solver in mixed precision are shown in figure 6.6. With 49.98% and 48.33% the solver and the rest of CalculiX require almost the same amount of time. Additionally, optimizations introduced in section 6.1 lead to faster matrix format conversion, which only amounts to 1.7% now instead of 6.9%.

Since the time spent within the equation solver is around 50% of the total time, the speedups that were presented in figure 6.4 and figure 6.5 almost halve when considering CalculiX's total computation time. This is displayed in figure 6.7. For our benchmark cases, the optimized software runs on average 3.1 times faster for pure CPU computation and 4.4 times for hybrid computation. The variance in performance for different input decks is very significant. Small or linear cases tend to achieve merely between 1 and 2 TFLOP/s during factorization and therefore barely perform twice as fast as the original implementation. Jobs that require a lot of operations, such as J5 and J9, produced up to 5 TFLOP/s and reach speedups up to 7.42.

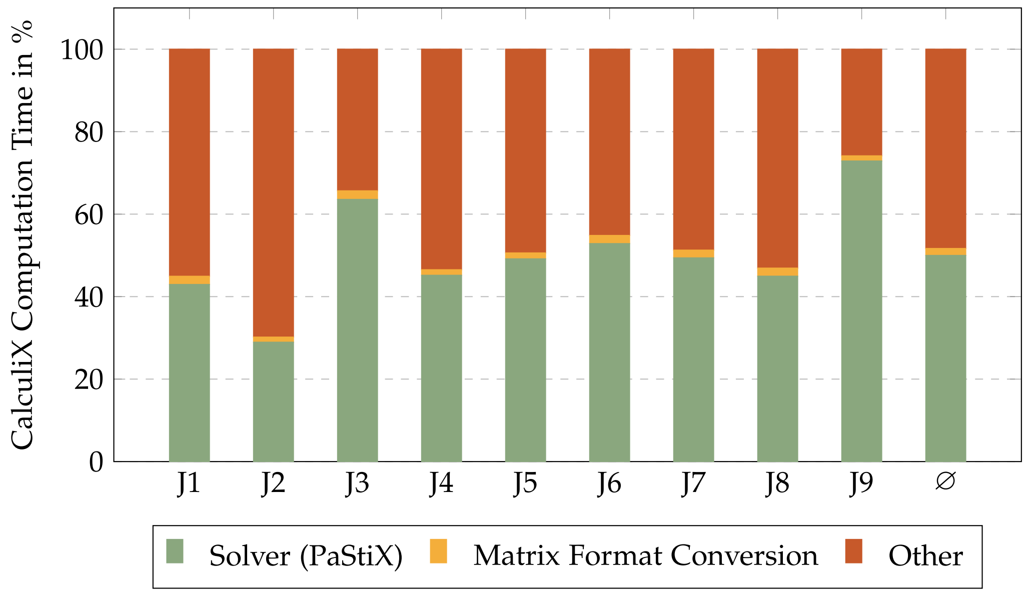

Figure 6.6: Structure of CalculiX computation time with PaStiXin hybrid and mixed precision mode. Executed on T1 with 8 threads.

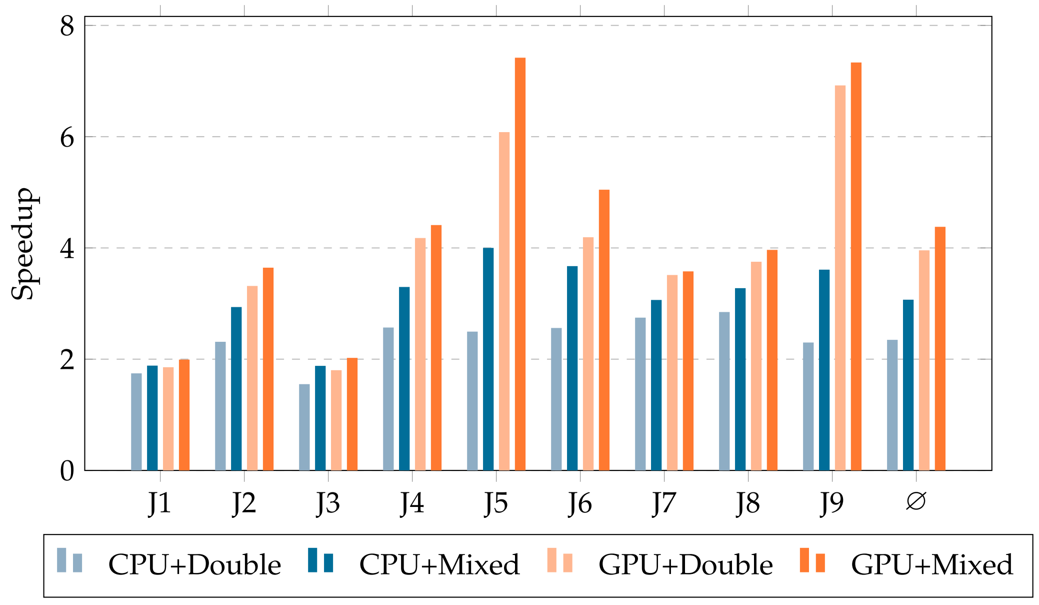

Figure 6.7: CalculiX performace comparison with PARDISO and PaStiX as solvers. The speedup is calculated with the original CalculiX version with PARDISO as a solver. Executed on T1 with 8 threads.

7 Conclusion

In this work we analyzed the options for possible GPU acceleration of linear equation solvers in the context of applications in structural mechanics. The goal was to accelerate the FEM simulation software CalculiX. We showed that classical iterative methods with block Jacobi or ILU preconditioning do not converge within reasonable time and are not an option that can be considered. In general, there are two approaches for the acceleration of direct solving methods. A pure GPU implementation is not feasible due to the fact that the entire matrix has to reside on the device memory and this is not possible for large geometries. Thus, a hybrid strategy, in which BLAS 3 operations with high computational intensity are offloaded to the GPU, was identified as the most promising approach. The software PaStiX implements this approach most convincingly by defining its algorithm for the task-based scheduling library PaRSEC. PaRSEC manages dependencies between tasks and schedules them to multiple CPU cores or GPUs.

By activating GPU offloading, PaStiX's factorization step gained on average a performance boost of 100% for our benchmark cases. By implementing pinned memory optimizations, we were able elevate this to 155%. This shows that the data transfer between CPU and GPU is the bottleneck of the hybrid strategy. Further optimization of GPU kernels will therefore not lead to significant total performance gain. Instead, we implemented a mixed precision feature that allows factorization to be executed in single precision. Since a very accurate result is required, iterative refinement is applied afterwards in double precision. PaStiX is memory-bound but this feature still succeeds because the `float` values transferred between main and device memory are half the size of the `double` values. Factorization in single precision runs 97% faster on the CPU and 72% faster using the hybrid mode. Including the additional iterative refinement, which we partially accelerated, total speedups of 1.46 and 1.2 are achieved for CPU and hybrid version.

Nevertheless, the overall performance suffered too much from the effects of Amdahl's law. The factorization and iterative refinement procedures are parallelized and optimized extensively but only accounted for 26.8% of the total execution time when deploying 8 threads and GPU. Massive overall speedup could further be achieved by optimizing and parallelizing the remaining 73.2%. First, we optimized the sparse matrix preprocessing which amounted to almost a quarter of the total computation time. Using 8 threads, it is now 7.3 times faster than the previously sequential implementation. Eventually, the reordering and analysis step were the remaining large sequential steps. Instead of parallelizing those, we modified the invocation of PaStiX inside CalculiX such that we possibly invoke it with the same sparsity pattern in subsequent iterations. This way we can reuse the reordering permutation and analysis data in 90% of the PaStiX invocations. This increases the ratio of numerical computation (factorizing, solving and refinement) to 66.1% of PaStiX's total computation time. Ultimately, the optimized hybrid equation solver returns a solution on average 7.0 times faster than the previous implementation that utilized the multi-threaded library PARDISO. Replacing the PARDISO solver in CalculiX by the optimized PaStiX implementation results in a speedup of 3.1 for the pure CPU mode and 4.4 when enabling GPU offloading.

7.1 Future Work

We performed several optimizations for PaStiX and with the current hardware there is only little room for further performance improvements. The LDLT implementation could be enhanced as currently only the `GEMM`s are offloaded to the GPU. As most of the matrices in structural mechanics are asymmetric and not eligible for LDLT, we did not prioritize this matter. Furthermore, with the introduction of mixed precision, more iterative refinement is required, which is computationally dominated by preconditioning. So far we determined that a reasonable execution of such on the GPU is not feasible because the *LU* matrices tend to be very large. Similar offloading strategies as for the factorization might allow partial computation on the GPU. This can potentially reduce the time required for iterative refinement.

Another problem is that CalculiX itself requires as much computation time as the equation solver. To keep up with PaStiX's performance, the rest of CalculiX needs to be optimized and preferably parallelized for GPGPU.

The most crucial change for PaStiX's performance will be the introduction of PCI Express 4.0 for GPUs. We noticed that additional memory bandwidth gained by the pinned memory optimization translated linearly into general performance increase. PCI Express 4.0 promises doubled bandwidth which will significantly boost the performance of PaStiX's GPU offloading algorithm.

List of Figures

2.1 1D Axially Loaded Bar 4
2.2 Node Partitioning of Axially Loaded Bar 6
2.3 Visualization of Block LU Procedure 15
2.4 Sparsity Pattern of a Matrix 18
2.5 Sparsity Pattern of a Matrix Reordered with the RCM Algorithm 19
2.6 Graph Partitioning for Nested Dissection 20
2.7 Sparsity Pattern of a Matrix Reordered with a Dissection Algorithm . . 21
2.8 Elimination Tree Generated by Nested Dissection 22

3.1 Structure of CalculiX's Computation Time with PARDISO 29
3.2 Convergence Behavior for GMRES 34
3.3 Baseline Comparison of PaStiX and PARDISO 35
3.4 Baseline Comparison of PaStiX's CPU and Hybrid Performance during Factorization 36
3.5 Tasks Associated to One Supernode 37
3.6 Structure of PaStiX's Computation Time in Hybrid Mode 38

4.1 Speedup by Performing the Factorization in Single Precision 40
4.2 Impact of Mixed Precision Implementation on CPU and Hybrid Solver 42

5.1 Comparison of SCOTCH and METIS 47
5.2 Performance Comparison of StarPU and PaRSEC 47
5.3 Effects of Splitting Parameters 48
5.4 Progression of Automatic Tuning with OpenTuner 49
5.5 Memory Bandwith with and without Pinned Memory 50
5.6 Performance for Allocation of Pinned Memory 51
5.7 Performance Impact of Pinned Memory on PaStiX's Factorization 51
5.8 Task Flow for Matrix Preprocessing 53
5.9 Performance Comparison Approaches for Computing Row Offsets . . . 55
5.10 Speedup for Shared Memory Parallelization of Matrix Preprocessing . . 56
5.11 Performance Comparison of the Matrix Preprocessing Step 57

5.12 Performance Comparison of CPU and GPU Implementation of GMRES in PaStiX without Preconditioning 60
5.13 Speedup for Hybrid Implementation of Right-Preconditioned GMRES . 60

6.1 Speedup when Being able to Reuse the Sparsity Pattern 62
6.2 Impact of Contact Approximation on Reusability and Factorization . . . 64
6.3 Amortized Structure of PaStiX's Computation Time including Optimizations 66
6.4 PaStiX's Performance in Comparison to PARDISO 66
6.5 GPU Speedup of PaStiX's Amortized Performance in CalculiX 67
6.6 Structure of CalculiX Computation Time with PaStiX 68
6.7 CalculiX Performace Comparison with PARDISO and PaStiX as Solvers 68

List of Tables

2.1 Matrix Storage in Coordinate Format 9
2.2 Matrix Storage in CSR Format 9
2.3 Matrix Storage in CSC Format 9
2.4 Memory Type Comparison 11
2.5 Triangular Decompositions 13

3.1 Matrices used for Benchmarking 28
3.2 Matrix Storage in CalculiX's Format 30

4.1 Required Iterations in Double and Mixed Precision 41
4.2 Required Iterations for Half Precision Approaches 44

5.1 Pinned Memory Allocation Measurements 52
5.2 Size and GPU Transfer Times of Sparse Benchmarking Matrices 59

6.1 Reusing Capability for the Benchmark Cases 63

Bibliography

[AG99] C. Ashcraft and R. G. Grimes. “SPOOLES: An Object-Oriented Sparse Matrix Library.” In: *PPSC*. 1999.

[Agu+09] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. “Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects.” In: *Journal of Physics: Conference Series*. Vol. 180. 1. IOP Publishing. 2009, p. 012037.

[Aja09] J. Ajanovic. “PCI express 3.0 overview.” In: *Proceedings of Hot Chip: A Symposium on High Performance Chips*. Vol. 69. 2009, p. 143.

[Ame+00] P. R. Amestoy, I. S. Duff, J.-Y. LExcellent, and J. Koster. “MUMPS: a general purpose distributed memory sparse solver.” In: *International Workshop on Applied Parallel Computing*. Springer. 2000, pp. 121–130.

[Ans+14] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe. “Opentuner: An extensible framework for program autotuning.” In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.

[Anz+19] H. Anzt, T. Cojean, G. Flegar, T. Grutzmacher, P. Nayak, and T. Ribizel. “An Automated Performance Evaluation Framework for the GINKGO Software Ecosystem.” In: *90th Annual Meeting of the International Associaten of Applied Mathematics and Mechanics, GAMM*. 2019.

[Aug+11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.” In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[Bab+09] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. “Accelerating scientific computations with mixed precision algorithms.” In: *Computer Physics Communications* 180.12 (2009), pp. 2526–2533.

[Ban07] Banerjee. *Axially loaded bar: The Finite Element Solution*. `https://en.wikiversity.org/wiki/Introduction_to_finite_elements/Axial_bar_finite_element_solution`. Accessed 30-March-2020. 2007.

[Bav16] E. T. Bavier. "Replicated Computational Results (RCR) Report for A Sparse Symmetric Indefinite Direct Solver for GPU Architectures." In: *ACM Transactions on Mathematical Software (TOMS)* 42.1 (2016), pp. 1–10.

[Bos+13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. "Parsec: Exploiting heterogeneity to enhance scalability." In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45.

[But+07] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. "Mixed precision iterative refinement techniques for the solution of dense linear systems." In: *The International Journal of High Performance Computing Applications* 21.4 (2007), pp. 457–466.

[CH17] E. Carson and N. J. Higham. "A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems." In: *SIAM Journal on Scientific Computing* 39.6 (2017), A2834–A2856.

[Cha20] Chatzi. *Lecture notes in Method of Finite Elements I*. Mar. 2020.

[Cor19a] I. Corporation. *Intel Xeon Gold 6244 Processor*. `https://ark.intel.com/content/www/us/en/ark/products/192442/intel-xeon-gold-6244-processor-24-75m-cache-3-60-ghz.html`. Accessed 30-March-2020. 2019.

[Cor19b] I. Corporation. *Intel Xeon Platinum 8280 Processor*. `https://ark.intel.com/content/www/us/en/ark/products/192478/intel-xeon-platinum-8280-processor-38-5m-cache-2-70-ghz.html`. Accessed 30-March-2020. 2019.

[CW+11] D. Y. Chenhan, W. Wang, et al. "A CPU–GPU hybrid approach for the unsymmetric multifrontal method." In: *Parallel Computing* 37.12 (2011), pp. 759–770.

[DER87] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. USA: Clarendon Press, 1987. ISBN: 0198534213.

[Dho17] G. Dhondt. "CalculiX CrunchiX users manual version 2.12." In: *URL http: www. dhondt. de/ccx* 2 (2017).

[DRS16] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar. "A survey of direct methods for sparse linear systems." In: *Acta Numerica* 25 (2016), pp. 383–566.

[DW98] G. Dhondt and K. Wittig. "Calculix: a free software three-dimensional structural finite element program." In: *MTU Aero Engines GmbH, Munich* (1998).

[Geo+11] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury. "Multifrontal factorization of sparse SPD matrices on GPUs." In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pp. 372–383.

[Hai+18] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers." In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 603–613.

[HM08] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era." In: *Computer* 41.7 (2008), pp. 33–38.

[HRR02] P. Hénon, P. Ramet, and J. Roman. "PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems." In: *Parallel Computing* 28.2 (2002), pp. 301–321.

[Huc17] Huckle. *Lecture notes in Parallel Numerics*. Oct. 2017.

[KK98] G. Karypis and V. Kumar. "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices." In: *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* 38 (1998).

[KP09] G. P. Krawezik and G. Poole. "Accelerating the ANSYS direct sparse solver with GPUs." In: *Symposium on Application Accelerators in High Performance Computing, SAAHPC*. 2009.

[Lac+14] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault. "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes." In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE. 2014, pp. 29–38.

[Lac15] X. Lacoste. "Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems." PhD thesis. 2015.

[Li05] X. S. Li. "An overview of SuperLU: Algorithms, implementation, and user interface." In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 302–325.

[LS15] Y. Liu and B. Schmidt. "LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs." In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2015, pp. 82–89.

[Luc+10] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes. "Multifrontal computations on GPUs and their multi-core hosts." In: *International Conference on High Performance Computing for Computational Science*. Springer. 2010, pp. 71–82.

[Mac11] C. A. Mack. "Fifty years of Moore's law." In: *IEEE Transactions on semiconductor manufacturing* 24.2 (2011), pp. 202–207.

[McC10] C. McClanahan. "History and evolution of gpu architecture." In: *A Survey Paper* 9 (2010).

[Nau+15] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, et al. "AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods." In: *SIAM Journal on Scientific Computing* 37.5 (2015), S602–S626.

[NVI17] T. NVIDIA. "V100 GPU architecture. the worlds most advanced data center GPU. Version WP-08608-001_v1. 1." In: *NVIDIA. Aug* (2017), p. 108.

[Oña] E. Oñate. "Structural Analysis with the Finite Element Method Linear Statics Volume 2. Beams, Plates and Shells." In: ().

[PB12] E. Peise and P. Bientinesi. "Performance modeling for dense linear algebra." In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 406–416.

[Pic+17] G. Pichon, M. Faverge, P. Ramet, and J. Roman. "Reordering strategy for blocking optimization in sparse linear solvers." In: *SIAM Journal on Matrix Analysis and Applications* 38.1 (2017), pp. 226–248.

[PR96] F. Pellegrini and J. Roman. "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs." In: *International Conference on High-Performance Computing and Networking*. Springer. 1996, pp. 493–498.

[Rus15] W. Rust. *Non-linear finite element analysis in structural mechanics*. Springer, 2015.

[Saa03] Y. Saad. *Iterative methods for sparse linear systems*. Vol. 82. siam, 2003.

[SG04] O. Schenk and K. Gärtner. "Solving unsymmetric sparse systems of linear equations with PARDISO." In: *Future Generation Computer Systems* 20.3 (2004), pp. 475–487.

[SHG04] J. A. Scott, Y. Hu, and N. I. Gould. "An evaluation of sparse direct symmetric solvers: an introduction and preliminary findings." In: *International Workshop on Applied Parallel Computing*. Springer. 2004, pp. 818–827.

[Shi19] A. Shilov. *Kingston Reveals DDR4-2933*. `https://www.anandtech.com/show/14162/kingston-reveals-ddr42933-registered-dimms-for-cascade-lakesp`. Accessed 30-March-2020. 2019.

[SVL14] P. Sao, R. Vuduc, and X. S. Li. "A distributed CPU-GPU sparse direct solver." In: *European Conference on Parallel Processing*. Springer. 2014, pp. 487–498.

[Wal16] M. M. Waldrop. "The chips are down for Moores law." In: *Nature News* 530.7589 (2016), p. 144.

[Wan+16] H. Wang, W. Liu, K. Hou, and W.-c. Feng. "Parallel transposition of sparse data structures." In: *Proceedings of the 2016 International Conference on Supercomputing*. 2016, pp. 1–13.

[Wat15] A. J. Wathen. "Preconditioning." In: *Acta Numerica* 24 (2015), pp. 329–376.

[Wil13] N. Wilt. *CUDA Handbook - A Comprehensive Guide to GPU Programming, The*. Amsterdam: Addison-Wesley, 2013. ISBN: 013-3-261-506-.