RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

utbm
université de technologie
Belfort-Montbéliard

UNIVERSITY OF TECHNOLOGY OF BELFORT-MONTBÉLIARD

Optimization of Eigensolvers with GPU

Internship Report ST50 - P2024

Gabriel Schwab
Informatique
Vision Artificielle

Ansys France
35-37 Rue Louis Guérin,
69100 Villeurbanne
https://www.ansys.com

Company tutor
Frédéric Thevenon

UTBM tutor
Jean-Charles Créput

Acknowledgements

First of all, I am very grateful to Frédéric Thevenon for giving me the opportunity to work on such an exciting topic. I also thank him for his trust and support throughout my internship. Many thanks to Chady Zaza and Jeff Beisheim from the MAPDL solver team for their collaboration and technical help.

I would also like to thank Florent Lopez, Guillaume Buron and Anthony Giacoma for the invaluable advice they gave me, as well as for the interesting conversations we had during these six months.

Finally, I thank my internship supervisor, Jean-Charles Créput, for his responsiveness and for answering all my questions.

Introduction

As part of my studies at the University of Technology of Belfort-Montbéliard, I completed my final internship at Ansys France in Villeurbanne, which focused on the GPU acceleration of an eigensolver.

Solvers are computational tools or algorithm used to find numerical solution of equation or system of equations. In particular, eigensolver finds eigenvalues and eigenvectors of a matrix, which are crucial in many scientific and engineering applications.

In recent years, GPUs have shown impressive capabilities in a variety of tasks, especially in scientific computing. The aim of this work is to identify computationally expensive operations and to test different algorithms and tools that could accelerate the solver using GPUs, while ensuring robust numerical results.

Contents

List of Figures

List of Algorithms

Acronyms

BLAS Basic Linear Algebra Subroutines. 11, 15, 20, 27, 38

CPU Central Processing Unit. 11

CSR Compressed Sparse Row. 9

dof degrees of freedom. 31, 32

FLOPS Floating Point Operations per Second. 27

GPU Graphical Processing Unit. 11

HPC High Performance Computing. 7

HPD Hermitian Positive Definite. 9

LAPACK Linear Algebra PACKage. 11, 22, 25

MKL Intel oneAPI Math Kernel Library. 27

nnz non-zero elements. 9, 10

QEP Quadratic Eigenvalue Problem. 17, 37

RRQR Rank Revealing QR. 39

SPD Symmetric Positive Definite. 9, 22

TOAR Two-level Orthogonal ARnoldi. 18, 37

Chapter 1

Company Overview

1.1 Ansys

Ansys is an American company based in Canonsburg, employing over 6000 people, with more than 600 in France spread across several sites. The company designs numerical simulation software sold to other businesses in various industries (aerospace, automotive, etc.) to simulate physical effects during product design. The numerous applications of this software have made Ansys a dominant player in the numerical simulation market, with many companies using their products. The main products of Ansys are:

- Ansys Mechanical for numerical simulation in mechanics
- Fluent for fluid dynamics (acquired in 2006)
- Ansoft for electromagnetics (acquired in 2008)

There are, of course, other software applications for various purposes such as optics and material databases.

1.2 MAPDL Solver Team

John A. Swanson founded Ansys in 1970. His first finite element solver, written in Fortran, enabled simulations in the field of structural mechanics. Since then, the company has expanded its portfolio of solvers and simulation applications across numerous domains. However, MAPDL, the original solver, remains one of the company's technological gems. It continues to evolve and expand its capabilities, thanks to an international development team.

In Lyon, part of this solver team is responsible for certain functionalities: eigenvalue analysis, linear dynamics applications, acoustic applications, generic HPC mathematical tools, and establishing a communication architecture around this solver.

Chapter 2

Background

2.1 Notation

Throughout this work, we usually follow the following convention for notating linear algebra:

- The fields of real and complex numbers are denoted by $\mathbb{R}$ and $\mathbb{C}$, respectively.
- Elements in $\mathbb{R}$ and $\mathbb{C}$, scalars, are denoted by lowercase letters, $a, b, c, ...$ or $\alpha, \beta, \gamma, ...$
- Vectors are denoted by boldface lowercase letters, $\mathbf{u}, \mathbf{v}, \mathbf{w}, ...$ or $\mathbf{x}, \mathbf{y}, \mathbf{z}, ...$
- Matrices are denoted by uppercase roman letters, $A, B, C, ...$
- A block matrix is a matrix that is defined using smaller matrices, called blocks.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

- To simplify complex indexing, we use the notation $1 : j = \{1, 2, ..., j\}$, thus

$$A_{1:j,k} = \begin{bmatrix} a_{1,k} \\ a_{2,k} \\ \vdots \\ a_{j,k} \end{bmatrix} \text{ denote the } k\text{-th column of } A \text{ of size } j$$

- Concerning Kyrlov methods, the resulting subspace matrices are denoted by caligraphic uppercase letters, whose subscript is denoting the subspace order, $\mathcal{Q}_k, \mathcal{K}_k, \mathcal{V}_k, ...$
- We write boldface $\mathbf{0}$ for a length n column vector of zeros.

We formulate all of the methods described in this report to be correct either in real or in complex arithmetic, using a single notation for both cases. We support this with the following notation:

- For a scalar α, $\overline{\alpha}$ denotes its complex conjugate if α is complex, else $\overline{\alpha} = \alpha$ if α is real.
- For a column vector x of length n, x^* denotes its transpose ($x^* = x^T$) if x is real, and its complex conjugate transpose ($x^* = \overline{x}^T$) if x is complex.
- Similarly, for an $m \times n$ matrix A, A^* denotes its transpose ($A^* = A^T$) if A is real, and its complex conjugate transpose ($A^* = \overline{A}^T$) if A is complex.
- For two vectors x and y of the same length n, we define their Euclidean inner product as $\langle x, y \rangle = y^* x$, whether the vectors are real or complex.

In the next sections, we will use the term "symmetric" to describe both real symmetric matrices ($A^T = A$), and complex Hermitian matrices ($A = \overline{A^T}$). Similarly, we say Symmetric Positive Definite (SPD) of both real symmetric positive definite matrices, and complex Hermitian Positive Definite (HPD) matrices.

Given an $n \times n$ SPD matrix M, the operation $\langle x, y \rangle_M = y^* M x$ defines an inner product, which we call the M *inner product*. We say analogously that the two n vectors x and y are M*orthogonal* ($x \perp_M y$), when $\langle x, y \rangle_M = 0$.

We distinguish the computed quantity in finite precision arithmetic with a hat. For instance we denote the computed version of the Q matrix by $\hat{Q}$ which is subject to numerical errors.

2.2 Sparse Linear Algebra

Sparse linear algebra deals with the manipulation and solution of linear algebraic problems where most of the elements in the matrices and vectors involved are zero. This is a common scenario in scientific computing. By contrast, if most of the elements are non-zero, the matrices or vectors are considered dense.

When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix.

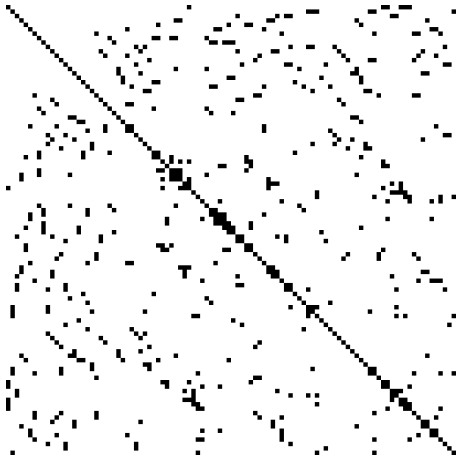

Figure 2.1: A sparse matrix obtained when solving a finite element problem in two dimensions. The non-zero elements are shown in black [34].

There is many formats for storing sparse matrices, we will only present one of them here since it's the only one we will use. This format is the **Compressed Sparse Row (CSR)**. In this format the row indices are compressed and replaced by an array of *offsets*. A CSR matrix is represented by the following parameters:

- The number of rows in the matrix.
- The number of columns in the matrix.
- The number of **non-zero elements (nnz)** in the matrix.

- The row offsets vector of length `number of rows + 1` that represents the starting position of each row in the columns and values vector.
- The column indices vector of length `nnz` that contains the column indices of the corresponding elements in the values vector.
- The values vector of length `nnz` that holds all nonzero values of the matrix.

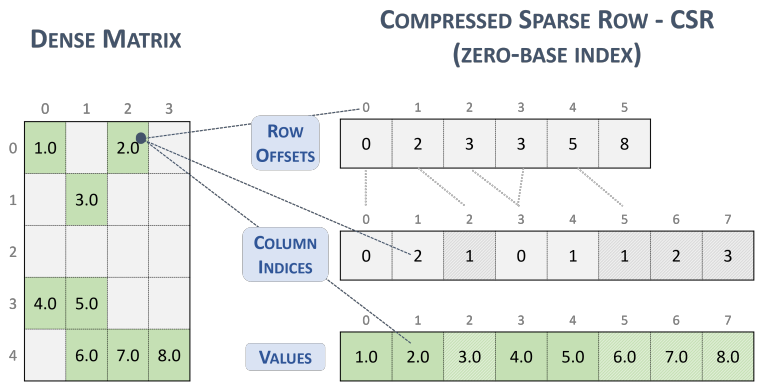

Figure 2.2: CSR matrix format [29].

2.3 Numerical Methods

Numerical analysis is the study of algorithms that use numerical approximation (as opposed to symbolic manipulations) for the problems of mathematical analysis. It is the study of numerical methods that attempt to find approximate solutions of problems rather than the exact ones.

2.3.1 Finite Precision Arithmetic

Numerical computing is based on the well-known floating-point representation of numbers, of the form

$$x = smb^e$$

where s is the sign of x, m its mantissa, b the basis (usually b=2) and e the exponent. Therefore, during computations, numbers are in general subject to *roundoff error*. The widely-used IEEE 754 [16] norm for representing floating point numbers and computing with them ensures that

$$\text{for all } x \in \mathbb{R}, \text{there exists } \varepsilon, |\varepsilon| < \varepsilon_{\text{mach}} \text{ such that } \text{fl}(x) = x(1+\varepsilon)$$

where $x \mapsto \text{fl}(x)$ is the operator converting a number to its floating point representation and $\varepsilon_{\text{mach}}$, ofter called "machine epsilon" or "machine precision", is the maximum modulus of relative round-off error.

Throughout this work, we essential focus on two floating point precision,

- 32-bit floating point (`float` in C++), whose $\varepsilon_{\text{mach}}$ = 1.19e-07
- 64-bit floating point (`double` in C++), whose $\varepsilon_{\text{mach}}$ = 2.22e-16

2.3.2 BLAS and LAPACK

Basic Linear Algebra Subroutines (BLAS) [22] is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.

BLAS functionality is categorized into three sets of routines called "levels", which correspond to both the chronological order of definition and publication, as well as the degree of the polynomial in the complexities of algorithms. Level 1 BLAS operations typically take linear time, $O(n)$, Level 2 operations quadratic time $O(n^2)$ and Level 3 operations cubic time $O(n^3)$.

Linear Algebra PACKage (LAPACK) [3] is a standard software library for numerical linear algebra. It provides routines for solving systems of linear equations and linear least squares, eigenvalue problems, and singular value decomposition.

When we program numerical linear algebra algorithms, we rely on these libraries to maximize the performances of the computations, since their implementations are often optimized for speed on a particular machine, so using them can bring substantial performance benefits. This is especially true when these operations are performed on a GPU. GPUs scale much better than CPUs as the size of the data increases [23]. Therefore, we typically prefer implementing algorithms that use matrix-vector or matrix-matrix operations when utilizing a GPU.

2.4 Graphical Processing Units

Graphical Processing Unit (GPU) is a specialized electronic circuit initially designed to accelerate computer graphics and image processing. Because basic numerical linear algebra operations play crucial roles in real time 3D computer graphics, GPUs are designed for this set of operations. Numerical linear algebra applications can achieve significantly higher performance when using GPUs, as they provide greater peak performance and bandwidth compared to multi-core CPUs.

The figure below illustrates the main differences in hardware architecture between CPUs and GPUs. The transistor counts associated with various functions are represented abstractly by the relative sizes of the different shaded areas.

Concerning the programming model of GPUs, we distinguish the *host* and the *device*. The host refers to the CPU and its memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernels which are functions executed on the device. These kernels are executed by many GPU threads in parallel. However, host cannot access directly memory allocated on the device and vice versa. Therefore, device computing often require additional memory transfers between host and devices.

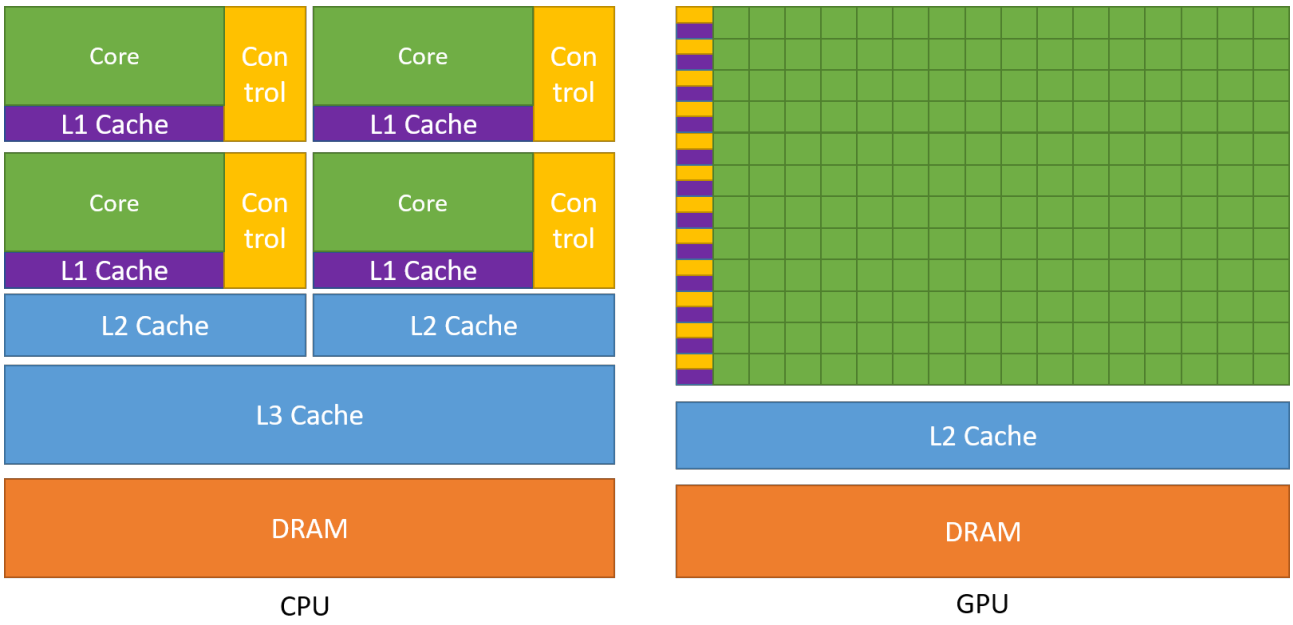

Figure 2.3: Comparing the relative capabilities of the basic elements of CPU and GPU architectures [27].

2.5 The Eigenvalue Problem

2.5.1 Definition

The **(standard) eigenvalue problem** is as follows.

Given a square matrix $A \in \mathbb{R}^{n \times n}$, find $\lambda \in \mathbb{C}$ and $\mathbf{x} \in \mathbb{C}^n, \mathbf{x} \neq \mathbf{0}$, such that

$$A\mathbf{x} = \lambda\mathbf{x} \tag{2.1}$$

or, equivalently, such that

$$(A - \lambda I)\mathbf{x} = \mathbf{0} \tag{2.2}$$

Let the pair $(\lambda, \mathbf{x})$ be a solution of (2.1) or (2.2). Then

- λ is called an **eigenvalue** of A
- $\mathbf{x}$ is called an **eigenvector** corresponding to λ
- $(\lambda, \mathbf{x})$ is called **eigenpair** of A
- The set $\sigma(A)$ of *all* eigenvalues of A is called **spectrum** of A

The **(generalized) eigenvalue problem** is as follows.

Given two square matrices $A, B \in \mathbb{R}^{n \times n}$, find $\lambda \in \mathbb{C}$ and $\mathbf{x} \in \mathbb{C}, \mathbf{x} \neq \mathbf{0}$, such that

$$A\mathbf{x} = \lambda B\mathbf{x}, \tag{2.3}$$

or, equivalently, such that

$$(A - \lambda B)\mathbf{x} = \mathbf{0} \tag{2.4}$$

2.5.2 Application

In structural engineering, modal analysis uses the overall mass and stiffness of a structure to find the various periods at which it will naturally resonate. This process helps identify the natural frequencies (resonant frequencies) and corresponding mode shapes, which are critical in understanding how the structure will respond to dynamic loads such as wind, earthquakes, or mechanical vibrations.

The undamped free vibration of structures result in a generalized linear eigenvalue problem

$$M\ddot{\mathbf{u}} + K\mathbf{u} = 0 \tag{2.5}$$

where the K and M matrices are the stiffness and mass matrices, respectively, $\mathbf{u}$ is the displacement vector, and $\ddot{\mathbf{u}}$ is the acceleration. The solutions of these problems are of the form

$$\mathbf{u} = \boldsymbol{\phi}\cos(\omega t) \tag{2.6}$$

Introducing $\lambda = \omega^2$, differentiating twice, and reordering results in the form of the typical generalized linear eigenvalue problem

$$(K - \lambda M)\boldsymbol{\phi} = 0 \tag{2.7}$$

Since the frequency range of interest in these problems is the lower end of the spectrum, it advisable to use the spectral transformation with σ a chosen *shift*,

$$\mu = \frac{1}{\lambda - \sigma} \tag{2.8}$$

This will change the problem into

$$(K - \sigma M)\boldsymbol{\phi} = \frac{1}{\mu}M\boldsymbol{\phi} \tag{2.9}$$

The eigenvectors are invariant under the spectral transformation, and the eigenvalues may be recovered as

$$\lambda = \frac{1}{\mu} + \sigma \tag{2.10}$$

2.6 Krylov subspace methods

Krylov subspace methods [20] have been ranked as the 3rd with the greatest influence on the development and practice of science and engineering in the 20th century [10]. Indeed it have become a very useful and popular tool for solving large sets of linear and nonlinear equations, and large eigenvalue problems. One of the reason for their popularity is their simplicity and their generality. These methods have been increasingly accepted as efficient and reliable alternative to the more expensive methods that are usually employed for solving dense problems. This trend is likely to accelerate as models are becoming more complex and give rise to larger and larger matrix problems.

The n-th **Krylov subspace** generated by $A \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ is the linear subspace spanned by the images of $\mathbf{b}$ under the first n powers of A, that is,

$$\mathcal{K}_n(A, \mathbf{b}) = \text{span}\{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, ..., A^{n-1}\mathbf{b}\} \tag{2.11}$$

2.7 Arnoldi Algorithm

The **Arnoldi iteration** [4] is an iterative eigenvalue algorithm. Arnoldi finds and approximation to the eigenvalues and eigenvectors of general matrices by constructing an orthonormal basis of the Krylov subspace, which makes it particularly useful when dealing with large sparse matrices.

An intuitive method for finding the largest eigenvalue of a given $m \times m$ matrix A is the power iteration [25]: starting with an arbitraty initial vector $\mathbf{b}$, calculate $A\mathbf{b}, A^2\mathbf{b}$,... normalizing the result after every application of the matrix A.

This sequence converges to the eigenvector corresponding to the eigenvalue with the largest absolute value $\boldsymbol{\lambda_1}$. However, much potentially useful computations is wasted by only using the final result, $A^{n-1}\mathbf{b}$. This suggests that instead, we store the intermediate results and form the so-called *Krylov matrix*:

$$\mathcal{K}_n = [\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, ..., A^{n-1}\mathbf{b}] \tag{2.12}$$

The columns of this matrix are not in general orthogonal, but we can extract an orthogonal basis, via a method such as Gram-Schmidt algorithm. The resulting set of vectors is thus an orthogonal basis of the Krylov subspace $\mathcal{K}_n(A, \mathbf{b})$. We may expect the vectors of this basis to span good approximations of the eigenvectors corresponding to the n largest eigenvalues.

Algorithm 1 Arnoldi Algorithm

Input: $A \in \mathbb{C}^{n\times n}$, an arbitrary vector $\mathbf{b} \in \mathbb{C}^n$
Output: $\mathcal{Q}_k = [\mathbf{q}_1, ..., \mathbf{q}_k]$ and $\mathcal{H}_k \in \mathbb{C}^{(k+1)\times k}$

$\mathbf{q_1} = \mathbf{b}/\|\mathbf{b}\|_2$
for $j = 1, ..., k-1$ **do**
  $\mathbf{w} = A\mathbf{q_j}$
  for $i = 1, ..., j$ **do** `# Gram-Schmidt orthogonalization`
    $h_{ij} = \mathbf{q}_i^*\mathbf{w}$
    $\mathbf{w} = \mathbf{w} - h_{ij}\mathbf{q}_i$
  end for
  $h_{j+1,j} = \|\mathbf{w}\|$
  $\mathbf{q}_{j+1} = \mathbf{w}/h_{j+1,j}$
end for

Let $\mathcal{Q}_k$ the unitary matrix whose columns $\mathbf{q}_1, ..., \mathbf{q}_k$ are the orthonormal basis for $\mathcal{K}_k(A, \mathbf{b})$ generated by the Arnoldi algorithm, and let $\mathcal{H}_k$ be the $(k+1) \times k$ upper Hessenberg matrix defined at the k-th stage of the algorithm. We remove that last row of the matrix $\mathcal{H}_k$, then these matrices satisfy

$$\mathcal{H}_k = \mathcal{Q}_k^* A \mathcal{Q}_k \tag{2.13}$$

If $k < n$, then $\mathcal{H}_k$ is a low-rank approximation to A and the eigenvalues of $\mathcal{H}_k$ may be used as approximations for the eigenvalues of A. The eigenvalues of $\mathcal{H}_k$ are called *Ritz eigenvalues*. Since $\mathcal{H}_n$ is a Hessenberg matrix of modest size, its eigenvalues can be computed efficiently, for instance with the QR algorithm [11].

2.8 Lanczos Algorithm

The Lanczos algorithm [21], is the Arnoldi algorithm specialized to the case where A is symmetric. However, we obtain some significant computational savings in this special case. Since $\mathcal{H}_k$ is both symmetric and Hessenberg, it is tridiagonal. This means that in the inner loop of the Arnoldi algorithm (1), the range 1 to j can be replaced by $j-1$ to j. Thus instead of the $j+1$-terms recurrence at step j, the Lanczos iteration involves just a 3-term recurrence. The result is that each step of the Lanczos iteration is much cheaper than the corresponding step of the Arnoldi iteration.

Let us write $\alpha_j = h_{j,j}$ and $\beta_j = h_{j+1,j}$ for $j = 1, ..., k$. Then $\mathcal{H}_k$ becomes

$$\mathcal{T}_k = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{k-1} \\ & & & \beta_{k-1} & \alpha_k \end{bmatrix}$$

Algorithm 2 Lanczos Algorithm

Input: A a $n \times n$ symmetric matrix, an arbitrary vector $\mathbf{b}$ of size n
Output: $\mathcal{Q}_k = [\mathbf{q}_1, ..., \mathbf{q}_k]$ and $\mathcal{T}_k$ the $k \times k$ tridiagonal matrix

1: $\mathbf{q_0} = \mathbf{0}, \quad \beta_0 = 0$
2: $\mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|_2 \quad Q_1 = [\mathbf{q}_1]$
3: **for** $j = 1, 2, ..., k-1$ **do**
4: $\quad \mathbf{w} = A\mathbf{q}_j$
5: $\quad \alpha_j = \mathbf{q}_j^T \mathbf{w}$
6: $\quad \mathbf{w} = \mathbf{w} - \beta_{j-1}\mathbf{q}_{j-1} - \alpha_j \mathbf{q}_j$
7: $\quad \beta_j = \|\mathbf{w}\|_2$
8: $\quad \mathbf{q}_{j+1} = \mathbf{w}/\beta_j \quad \mathcal{Q}_{j+1} = [\mathcal{Q}_j \ \mathbf{q}_{j+1}]$
9: **end for**

Some modifications, re-arrangements or additional steps may be applied to this algorithm for reasons of numerical stability in a finite arithmetic context. We won't detail them here but it can be find in the literature such as in [13].

In context of large sparse matrices, *block* methods can offers advantages over classical methods. By performing operations on blocks of vectors instead of on single vector, we can take advantage of BLAS 3 routines instead of BLAS 2, and therefore speed-up the computation time. For iterative eigensolvers, block methods reduce the number of iterations required to resolve a cluster of nearby eigenvalues. For some applications, block methods are necessary for computing all the desired eigenvalues to the desired accuracy; they are not (just) a performance optimization.

The block Lanczos algorithm is similar to the classical Lanczos algorithm, the Krylov subspace is formed by the column vectors of the $n \times ks$ matrix $\mathcal{Q}_k = [Q_1, ..., Q_k]$, where s is the block size. Here is a block version of the Lanczos algorithm. [14].

Algorithm 3 Block Lanczos Algorithm

Input: H a $n \times n$ symmetric matrix, an arbitrary matrix Q_1 such as $Q_1^* Q_1 = I_s$
Output: $\mathcal{Q}_k = [Q_1, ..., Q_k]$ and $\mathcal{T}_k$ the $ks \times ks$ block tridiagonal matrix

1: $Q_0 = \mathbf{0}, \quad B_0 = \mathbf{0}$
2: **for** $j = 1, 2, ..., k-1$ **do**
3: $\quad W = HQ_j$
4: $\quad A_j = Q_j^T W$
5: $\quad W = W - Q_{j-1} B_{j-1}^T - Q_j A_j$
6: $\quad Q_{j+1}, B_j = \texttt{QR}(W)$ `# ` Q_{j+1} `is orthogonal and` B_j `is upper triangular`
7: **end for**

The matrices Q_j and W for $j = 1, 2, ..k$ are $n \times s$, whereas A_j and B_j are $p \times p$, with A_j symmetric. The algorithm also defines a $ks \times ks$ block tridiagonal matrix $\mathcal{T}_k$

$$\mathcal{T}_k = \begin{bmatrix} A_1 & B_1 & & & \\ B_1 & A_2 & B_2 & & \\ & B_2 & A_3 & \ddots & \\ & & \ddots & \ddots & B_{k-1} \\ & & & B_{k-1} & A_k \end{bmatrix}$$

The `QR` subroutines used in the algorithm compute the QR factorization of the matrix W. This operation factorize a $m \times n$ matrix A into a product $A = QR$ of a orthogonal $m \times m$ matrix Q and a upper triangular $n \times n$ matrix R. There are several methods for actually computing the QR decomposition, such as the Gram–Schmidt process, Householder transformations[15], or Givens rotations. Each has a number of advantages and disadvantages.

The next step is to consider the generalized symmetric eigenproblem (2.9). We finally obtain the Shifted Block Lanczos algorithm [14].

Algorithm 4 Shifted Block Lanczos Algorithm

Input: H a $n \times n$ symmetric matrix, an arbitrary matrix Q_1 such as $Q_1^T M Q_1 = I_s$
Output: $\mathcal{Q}_k = [Q_1, ..., Q_k]$ and T_k the $ks \times ks$ block tridiagonal matrix

1: $Q_0 = \mathbf{0}, \quad B_0 = \mathbf{0}$
2: **for** $j = 1, 2, ..., k-1$ **do**
3: $\quad W = (K - \sigma M)^{-1}(MQ_j)$
4: $\quad A_j = W^T(MQ_j)$
5: $\quad W = W - Q_{j-1} B_{j-1}^T - Q_j A_j$
6: $\quad Q_{j+1}, B_j = \texttt{M-QR}(W)$ `# ` Q_{j+1} `is M-orthogonal and` B_j `is upper triangular`
7: **end for**

The `M-QR` routine computes also the QR factorization of the matrix W as explained for the previous algorithm (3) but instead of computing Q an orthogonal matrix, it computes Q as a M-orthogonal matrix (i.e. $Q^T M Q = I$).

This algorithm can capture the eigenvalues around the given shift σ. Sometimes, the algorithm struggle to find all the eigenvalues close to the given shift after a lot of iterations. One strategy is to restart the algorithm with a different shift σ, to capture additional nearby eigenvalues.

2.9 Quadratic Eigenvalue Problem

2.9.1 Definition

The **Quadratic Eigenvalue Problem (QEP)** is as follows.

Given $M, C, K \in \mathbb{C}^{n\times n}$, find $\lambda \in \mathbb{C}$, left and right nonzero eigenvectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^n$, such that

$$(\lambda^2 M + \lambda C + K)\mathbf{x} = 0, \quad \mathbf{y}^*(\lambda^2 M + \lambda C + K) = 0 \tag{2.14}$$

2.9.2 Application

In the section 2.5.2, we described the application on the case of modal analysis and undamped free vibration of structures. The consideration of damping makes the analysis closer to the reality. The damping effect is represented by the presence of the damping matrix.

The quadratic eigenvalue problems result from the differential equation

$$M\ddot{\mathbf{u}} + C\dot{\mathbf{u}} + K\mathbf{u} = 0 \tag{2.15}$$

where C is the damping matrix, $\dot{\mathbf{u}}$ refers to the velocity. The solution of this homogeneous system is of the form

$$\mathbf{u} = e^{\lambda t}\boldsymbol{\sigma} \tag{2.16}$$

where $\boldsymbol{\sigma}$ is a complex vector and the λ eigenvalue is also complex in general. Substitution of the last equation and its time derivatives gives the characteristic equation

$$(\lambda^2 M + \lambda C + K)\boldsymbol{\Phi} = 0 \tag{2.17}$$

The QEP has a wide range of applications beyond structural analysis. Tisseur and Meerbergen [33] provide a excellent survey on the QEP and its various applications.

Two approaches are well-known to solve quadratic eigenvalue problems and quadratic system of equations. One approach finds an appropriate linearization that results in linear eigenvalue problems or a linear system of equations. Another approach projects larger sparse quadratic problem onto a lower dimensional subspace, and subsequently produce a small, dense QEP or a system of equations. The first approach has a drawback that it increases the condition number due to linearization that double the size of the problem.

The second approach can be summarize in the following steps:

1. Set $A = -K^{-1}C$ and $B = -K^{-1}M$ and use an algorithm to construct a lower dimensional subspace matrix $\mathcal{Q}$.
2. Compute reduced order coefficient matrices $M_m = \mathcal{Q}^T M \mathcal{Q}$, $C_m = \mathcal{Q}^T C \mathcal{Q}$ and $K_m = \mathcal{Q}^T K \mathcal{Q}$.
3. Use dense QEP solver to solve $(\theta^2 M_m + \theta C_m + K_m)\mathbf{g} = 0$ for Ritz paris $(\theta_i, \mathbf{g}_i)$.
4. Transform Ritz vectors $\mathbf{g}_i$ into full-order vectors $\boldsymbol{\Phi}_i = \mathcal{Q}\mathbf{g}_i$ and normalize it.

Note that we never explicitly compute the A and B matrices by inverting K. Usually, we only need the result of the matrix-vector product $\mathbf{x} = A\mathbf{r}$. We can implicitly calculate the result $\mathbf{x}$ by solving the linear system

$$\mathbf{x} = A\mathbf{r} \tag{2.18}$$
$$\mathbf{x} = -K^{-1}C\mathbf{r} \tag{2.19}$$
$$K\mathbf{x} = -C\mathbf{r} \tag{2.20}$$

Which in the end only implies two matrix-vector products and a linear solve.

We will use the state-of-the-art algorithm **Two-level Orthogonal ARnoldi (TOAR)** [24] for solving such QEPs by constructing reduced order problem.

Chapter 3

Work Done

In general, there are two main approaches to accelerating an application using a GPU. The first approach involves rewriting the application so that all computations are performed on the GPU. The second approach is a hybrid method that offloads specific operations to the GPU, targeting those that are likely to be faster on the GPU. Both methods have their own advantages and disadvantages.

The advantage of the first approach is that it fully exploits the GPU's potential, with no latency from memory transfers between the host and the device. On the other hand, the hybrid method can be more easily integrated into existing code and allows for focusing on highly parallelizable operations. However, this approach involves additional memory transfers, which might cause overhead.

In our case, we chose the hybrid approach which is obviously the easier to start with. We identified 3 main computational costs in the Shifted Block Lanczos algorithm (4).

- The matrix-vector products (line 3)
- The resolution of linear systems (line 3)
- The orthogonalization procedures (line 5)

We decided to address first the orthogonalization procedure, because the matrix-vector product acceleration was already in progress, and the resolution of linear systems was harder to begin with.

3.1 Orthogonalization Procedures

The goal of this work is to experiment with various orthogonalization procedures and explore whether they can be accelerated using the GPU. This involves considering two key aspects: performance and numerical stability. Our primary focus is on performance, but we must also ensure a minimum level of stability.

An orthogonalization procedure is a procedure that, given a set of vectors $S = [\mathbf{v}_n, ..., \mathbf{v}_n]$, generate an orthogonal set $S' = [\mathbf{u}_1, ..., \mathbf{u}_n]$ that spans the same n-dimensional subspace.

We will try different algorithms and measured their performances as well as their numerical stability in order to find the best one for our use case. Here is a table of all the acronyms of the studied algorithms.

CGS	Classical Gram-Schmidt
MGS	Modified Gram-Schmidt
CGS2	Classical Gram-Schmidt with reorthogonalization
BCGS	Block Classical Gram-Schmidt
BMGS	Block Modified Gram-Schmidt
BCGS2	Block Classical Gram-Schmidt with reorthogonalization
CholQR	Cholesky QR

Table 3.1: Acronyms for orthogonalization algorithm

3.1.1 Gram-Schmidt Algorithm and its Variants

Gram-Schmidt Algorithm is a well known way of finding a set of two or more vectors that are perpendicular to each other.

Algorithm 5 `CGS`

Input: $X = [\mathbf{x}_1, ..., \mathbf{x}_n] \in \mathbb{R}^{m \times n}$
Output: $Q = [\mathbf{q}_1, ..., \mathbf{q}_n] \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^TQ = I_n$

1: **for** $i = 1, ..., n$ **do**
2: $\quad \mathbf{r}_i = \mathbf{x}_i$
3: $\quad$ **for** $j = 1, ..., i-1$ **do**
4: $\quad\quad \mathbf{r}_i = \mathbf{r}_i - (\mathbf{q}_j^T \mathbf{x}_i)\mathbf{q}_j$
5: $\quad$ **end for**
6: $\quad \mathbf{q}_i = \mathbf{r}_i / \|\mathbf{r}_i\|_2$
7: **end for**

We can rewrite this algorithm in matrix notation to take advantages of the speed-up of matrix-vector product over vector-vector dot product. This allows us to leverage BLAS 2 operations. This version is even more interesting on GPU because it leverages matrix-vector product parallelism.

Algorithm 6 `CGS` (matrix notation)

Input: $X = [\mathbf{x}_1, ..., \mathbf{x}_n] \in \mathbb{R}^{m \times n}$
Output: $Q = [\mathbf{q}_1, ..., \mathbf{q}_n] \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^TQ = I_n$

1: $Q = X$
2: **for** $i = 1, ..., n$ **do**
3: $\quad R_{1:i-1,i} = Q_{:,1:i-1}^T X_{:,k}$
4: $\quad Q_{:,k} = X_{:,k} - Q_{:,1:k-1} R_{1:k-1,k}$
5: $\quad R_{k,k} = \|Q_{:,k}\|$
6: $\quad Q_{:,k} = Q_{:,k} / R_{k,k}$
7: **end for**

It turns out that the Gram-Schmidt procedure we introduced previously suffers from numerical instability: Round-off errors can accumulate and destroy orthogonality of the resulting vectors [19]. We introduce the modified Gram-Schmidt procedure to help remedy this issue.

Algorithm 7 `MGS`

Input: $X = [\mathbf{x}_1, ..., \mathbf{x}_n] \in \mathbb{R}^{m \times n}$
Output: $Q = [\mathbf{q}_1, ..., \mathbf{q}_n] \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^T Q = I_n$

for $i = 1, ..., n$ **do**
  $\mathbf{r}_i = \mathbf{x}_i$
  for $j = 1, ..., i-1$ **do**
    $\mathbf{r}_i = \mathbf{r}_i - (\mathbf{q}_j^T \mathbf{r}_i)\mathbf{q}_j$
  end for
  $\mathbf{q}_i = \mathbf{r}_i / \|\mathbf{r}_i\|_2$
end for

In classical Gram-Schmidt (CGS) we compute the projections of the **original** k-th column vector $\mathbf{x}_k$ onto all left orthogonal column vectors already computed $\{\mathbf{q}_j\}_{j=1}^{k-1}$. Then subtract those projections from $\mathbf{x}_k$:

$$\mathbf{r}_k = (I - Q_{k-1} Q_{k-1}^T)\mathbf{x}_k \tag{3.1}$$

In modified Gram–Schmidt (MGS) we compute the projection of the **computed** k-th column vector $\mathbf{x}_k$ onto $\mathbf{q}_1$ and then subtract this projection from $\mathbf{x}_k$. Then do the same sequentially for all $\{\mathbf{q}_j\}_{j=1}^{k-1}$:

$$\mathbf{r}_k = (I - \mathbf{q}_k \mathbf{q}_k^T)...(I - \mathbf{q}_1 \mathbf{q}_1^T)\mathbf{x}_k \tag{3.2}$$

See the difference highlighted in red in (5) and (7).

There is an even more numerically stable version of the Gram-Schmidt algorithm which is the classical Gram-Schmidt algorithm with reorthogonalization (CGS2). By repeating the process of orthogonalization twice, we ensure that orthogonality of the set of vectors only depends on the machine precision. This is also known as the Kahan-Parlett "twice-is-enough" algorithm [30].

Algorithm 8 `CGS2`

Input: $X \in \mathbb{R}^{m \times n}$
Output: $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^T Q = I_n$

$Q = X$
for $i = 1, ..., n$ **do**
  $R^{(1)} = Q_{:,1:i-1}^T X_{:,k}$ `# 1st CGS step`
  $Q^{(1)} = X_{:,k} - Q_{:,1:k-1} R^{(1)}$
  $R^{(2)} = Q_{:,1:i-1}^T Q^{(1)}$ `# 2nd CGS step`
  $Q_{:,k} = Q^{(1)} - Q_{:,1:k-1} R^{(2)}$
  $R_{k,k} = \|Q_{:,k}\|$
  $Q_{:,k} = Q_{:,k} / R_{k,k}$
  $R_{1:i-1,i} = R^{(1)} + R^{(2)}$
end for

3.1.2 Cholesky QR Factorization

The Cholesky QR factorization, or `CholQR` is also another algorithm that could be interesting because it doesn't require to work column by column sequentially (which is a bottleneck for

GPU acceleration). This algorithm only rely on 3 kernels calls, that are all part of the LAPACK library :

- `SYRK`, compute $A = X^T X$ which makes A matrix SPD
- `CHOL`, compute R with Cholesky factorization $RR^T = \texttt{Chol}(A)$
- `TRSM`, compute $Q = AR^{-1}$ by solving the triangular matrix equation $XR = A$

However, this algorithm is not suited for all sizes of matrices because the $X^T X$ operation squares the condition number of the matrix A.

Algorithm 9 `CholQR`

Input: $X \in \mathbb{R}^{m \times n}$
Output: $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^T Q = I_n$
1: $A = X^T X$
2: $R = \texttt{Chol(A)}$ `# Cholesky factorization`
3: $Q = AR^{-1}$

3.1.3 Block Gram-Schmidt Variants

As the orthogonalization procedures we are studying are part of a block version of the Lanczos algorithm (4), we also investigate block version of the standard Gram-Schmidt algorithm introduced in the previous section. For the same reason, we only consider algorithms that work left-to-right i.e., only information from the k first blocks of Q and X is necessary for generating Q_{k+1}.

This work is essentially based on the work of Carson et al.[8] which provides a great comprehensive categorization of block Gram-Schmidt algorithms (BGS), particularly those used in Krylov subspace methods to build orthonormal bases one block of vectors at a time.

In the following, we consider a matrix $X \in \mathbb{R}^{m \times n}, m \gg n$ which is partitioned into a set of p *block vectors*, each of size $m \times s$, i.e.

$$X = [X_1, X_2, ..., X_p]$$

We also consider the QR factorization $X = QR$, where $Q = [Q_1, ..., Q_p] \in \mathbb{R}^{m \times n}$ has the same structure than X and $R \in \mathbb{R}^{n \times n}$. (To remember the meaning of subscript, note that $m > n > p > s$ which is in the alphabetical order and $n = ps$).

We can decompose a BGS in two main part, the *intra*-block orthogonalization and the *inter*-block orthogonalization. We need to orthogonalize every block to each others (`InterOrtho`) and every vector within a block to each other (`IntraOrtho`). The *inter*-block orthogonalization algorithm is independent of the *intra*-block orthogonalization algorithm, which means we can mix any `InterOrtho` with any `IntraOrtho`.

The `IntraOrtho` routines have been presented in the previous section. We will now describe the `InterOrtho` ones independently. For simpler notation, we denote $A_{1:k}$ the sub matrix $A' = [A_1, ..., A_k]$.

Algorithm 10 `BCGS`

Input: $X \in \mathbb{R}^{m \times n}$
Output: $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^T Q = I_n$

$Q_1, R_{1,1} = \texttt{IntraOrtho}(X_1)$
for $i = 1, ..., p-1$ **do**
    $R_{1:i,i+1} = Q_{1:i}^T X_{i+1}$
    $W = X_{i+1} - Q_{1:i} R_{1:i,i+1}$
    $Q_{i+1}, R_{i+1,i+1} = \texttt{IntraOrtho}(W)$
end for

Algorithm 11 `BMGS`

Input: $X \in \mathbb{R}^{m \times n}$
Output: $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^T Q = I_n$

$Q_1, R_{1,1} = \texttt{IntraOrtho}(X_1)$
for $i = 1, ..., p-1$ **do**
    $W = X_{i+1}$
    for $j = 1, ..., i-1$ **do**
        $R_{j,i+1} = Q_j^T W$
        $W = W - Q_j R_{j,i+1}$
    end for
    $Q_{i+1}, R_{i+1,i+1} = \texttt{IntraOrtho}(W)$
end for

Algorithm 12 `BCGS2`

Input: $X \in \mathbb{R}^{m \times n}$
Output: $Q \in \mathbb{R}^{m \times n}, R \in \mathbb{R}^{n \times n}$ such that $QR = X$ and $Q^T Q = I_n$

$Q_1, R_{1,1} = \texttt{IntraOrtho}(X_1)$
for $i = 1, ..., p-1$ **do**
    $R^{(1)}_{1:i,i+1} = Q_{1:i}^T X_{i+1}$ `# 1st BCGS step`
    $W = X_{i+1} - Q_{1:i} R^{(1)}_{1:i,i+1}$
    $Q^{(1)}, R^{(1)}_{i+1,i+1} = \texttt{IntraOrtho}(W)$
    $R^{(2)}_{1:i,i+1} = Q_{1:i}^T Q^{(1)}$ `# 2nd BCGS step`
    $W = Q^{(1)} - Q_{1:i} R^{(2)}_{1:i,i+1}$
    $Q_{i+1}, R^{(2)}_{i+1,i+1} = \texttt{IntraOrtho}(W)$
    $R_{1:i,i+1} = R^{(1)}_{1:i,i+1} + R^{(2)}_{1:i,i+1} R^{(1)}_{i+1,i+1}$
    $R_{i+1,i+1} = R^{(2)}_{i+1,i+1} R^{(1)}_{i+1,i+1}$
end for

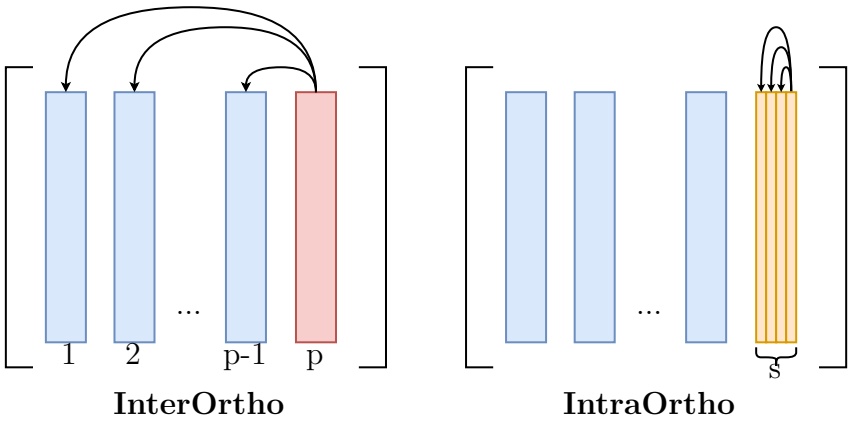

Figure 3.1: Block Orthogonalization Diagram

3.1.4 Numerical Properties

All of this orthogonalization procedures have different numerical properties. Since we use these algorithms to orthogonalize matrices, we are interested in a upper bounds on the loss of orthogonality due to round off errors during computations. Let $\hat{Q}$ be the result of an orthogonalization of X. We measure the loss of orthogonality as the quantity

$$\|I_n - \hat{Q}^T\hat{Q}\| \tag{3.3}$$

for some norm $\|\cdot\|$. We take the Euclidean norm, unless otherwise noted. We express the upper bound of (3.3) in terms of the machine precision $O(\varepsilon)$ and the matrix condition number $\kappa(X)$ which is the ratio between its largest and smallest singular value.

The following table summarize the upper bound on the loss of orthogonality after intra-block orthogonalization.

Algorithm	$\|I_n - \hat{Q}^T\hat{Q}\|$	Assumption on $\kappa(X)$	References
`CGS`	$O(\varepsilon)\kappa^{n-1}(X)$	$O(\varepsilon)\kappa(X) < 1$	[19]
`CholQR`	$O(\varepsilon)\kappa^2(X)$	$O(\varepsilon)\kappa^2(X) < 1$	[36]
`MGS`	$O(\varepsilon)\kappa(X)$	$O(\varepsilon)\kappa(X) < 1$	[6]
`CGS2`	$O(\varepsilon)$	$O(\varepsilon)\kappa(X) < 1$	[12],[1],[5]

Figure 3.2: Upper bound on loss of orthogonality for standard orthogonalization algorithms.

We notice that `CGS` is really unstable, its therefore a bad choice for most of the applications. `CGS2` doesn't depend on the matrix condition number, which is really interesting even though the computational cost is almost twice of the other ones.

Concerning the BGS, the bound on loss of orthogonality is depending on the bound of the `IntraOrtho` used. The following table summarize this bounding.

Knowing those theoretical bounding, we will try to reproduce these results and explore the behaviour of `InterOrtho` with different `IntraOrtho`. We denote this composition with the symbol $\circ$. For example, `BCGS` $\circ$ `CGS` is the Block Classical Gram-Schmidt algorithm that uses classical Gram-Schmidt algorithm as `IntraOrtho` routine.

Algorithm	`IntraOrtho` bound	$\|I_n - \hat{Q}^T\hat{Q}\|$	Assumption on $\kappa(X)$	References
`BCGS`	$O(\varepsilon)$	$O(\varepsilon)\kappa^{n-1}(X)$	$O(\varepsilon)\kappa(X) < 1$	conjectured in [8]
`BMGS`	$O(\varepsilon)\kappa(X)$	$O(\varepsilon)\kappa^2(X)$	$O(\varepsilon)\kappa(X) < 1$	[18] [8]
`BMGS`	$O(\varepsilon)$	$O(\varepsilon)\kappa(X)$	$O(\varepsilon)\kappa(X) < 1$	[18] [8]
`BCGS2`	$O(\varepsilon)$	$O(\varepsilon)$	$O(\varepsilon)\kappa(X) < 1$	[5]

Figure 3.3: Upper bound on loss of orthogonality for block orthogonalization algorithms.

3.1.5 Numerical Benchmarks

The goal is to measure the loss of orthogonality against the condition number of the matrix. We will carry out our experiments exclusively in double precision, using `float64` in other words. Therefore the machine precision is $\varepsilon =$ 1e-16. This is because in real-life use cases, the orthogonality is too sensible and critical to use single precision.

To generate matrices with a specific condition number $\kappa(X)$, we generate a $m \times n$ matrix

$$X = U\Sigma V^T$$

where $U, V \in \mathbb{R}^{m\times n}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{n\times n}$ is a diagonal matrices whose entries are such that

$$\kappa(X) = \frac{\sigma_{max}(\Sigma)}{\sigma_{min}(\Sigma)}$$

For this we use the `LATMS` routine from LAPACK.

The figure 3.4 shows us the result of the measurements. We can see that they are consistent with the theoretical results in 3.15. `CGS2` is very interesting because the loss of orthogonality doesn't depend on the condition number of the matrix, which makes it a better solution for badly conditioned matrices. In general, `MGS` is sufficient for common matrices.

The `CholQR` algorithm could have been a good alternative to Gram-Schmidt even if it's bounded by the square of the condition number. This is because we are not using big block sizes in the Lanczos algorithm, therefore we could expect a low condition number for each block. Nevertheless, after some preliminaries results on `CholQR` performances, we decided to not continue with this algorithm as it was much slower than Gram-Schmidt algorithms on GPU. For an orthogonalization of a block of 6 vectors of size 1e6, the `CholQR` was 100 times slower. This algorithm might be interesting on bigger block sizes though, but in our case, we were contrainted to small block sizes.

The figure 3.5 is also consistent with the theoretical bounds given in 3.3. We learn that the choice of the `IntraOrtho` can make a big difference depending on the `InterOrtho`. First, `BCGS` is so unstable, no matter what `IntraOrtho` is used, that we won't consider it as an option for a real use case. Thus, we can consider only `MGS` $\circ$ `CG2`, `BCGS2` $\circ$ `MGS` or `BCGS2` $\circ$ `CGS2`. The choice between this options will depends on the performances of the algorithms and the type of matrices to deal with.

There are also other Gram-Schmidt variants that use optional reorthogonalization. Those variants are really interesting for a CPU implementation, but since we target a GPU implementation, we won't explore this options for performance reasons.

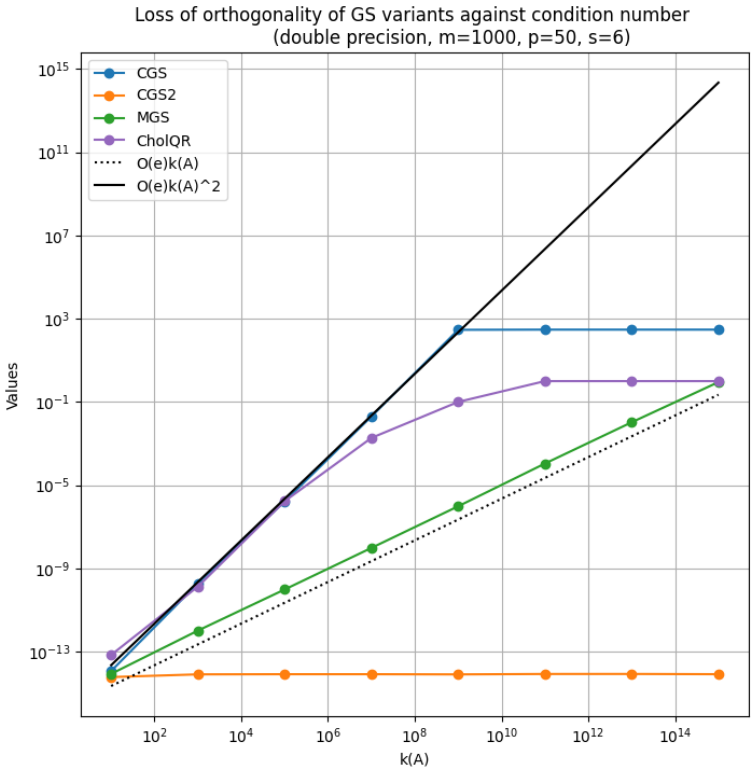

Figure 3.4: Loss of orthogonality measurement of GS variants

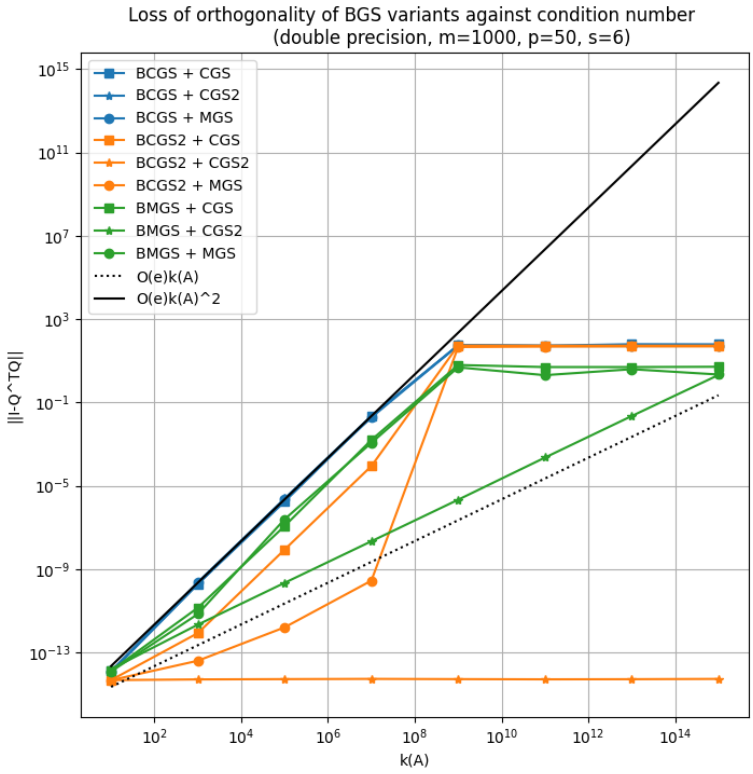

Figure 3.5: Loss of orthogonality measurement of BGS variants

3.1.6 Performance Benchmarks

The aim of this work is to compare the performances of the CPU implementation versus the GPU implementation of the algorithms shown above. For the CPU implementation, we used the Intel oneAPI Math Kernel Library (MKL) [17] for the BLAS kernels calls. For the GPU implementation, we focused on NVIDIA GPUs since they were the most accessible in the company and the most efficient on the market. So, we used the CUDA programming language as well as the cuBLAS library [26] which is the NVIDIA implementation of the BLAS standard.

Different metrics are relevant regarding the performance of an algorithm. We decided to measure the **Floating Point Operations per Second (FLOPS)**, which is the computational power of hardware, such as CPUs or GPUs. It tells us how many floating-point operations the hardware can perform every second. To measure the FLOPS of each algorithm, we divide their flop counts by their execution time. For the GPU implementation, we don't measure the time of data transfers between host and device.

The following table 3.6 summarizes the number of flops for basic linear algebra operations [7] and for GS algorithms. For the GS algorithms, we only give an approximation of the flops count for simplicity. For instance, the flops count for `CGS` is

$$(n-1)\frac{(n-2)}{2}4m + n(3m+1) = 2mn^2 - 3mn + 4m + n \approx 2mn^2$$

Regarding the block GS algorithms, we ignore the number of flops of the `IntraOrtho` since it is negligible compare to the whole algorithm. For instance in case of `BMGS` $\circ$ `Intra`, the flops count is

$$\begin{aligned} flops(\texttt{BMGS} \circ \texttt{Intra}) &= flops(\texttt{Intra}) + (p-2)\frac{(p-3)}{2}4ms^2 + (p-1)flops(\texttt{Intra}) \\ &\approx 2mp^2s^2 + pflops(\texttt{Intra}) \\ &\approx 2mp^2s^2 + pms^2 \\ &\approx 2mp^2s^2 \\ &\approx 2mn^2, (\text{ since } n = ps) \end{aligned}$$

As a result, the block version of GS is in the same order of magnitude than the classical ones.

Operation	Dimension	Flops
$\alpha = \mathbf{x}^T\mathbf{y}$	$\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$	$2n$
$\mathbf{y} = \mathbf{y} + A\mathbf{x}$	$A \in \mathbb{R}^{m\times n}, x \in \mathbb{R}^n, y \in \mathbb{R}^m$	$2mn$
$C = C + AB$	$A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}, C \in \mathbb{R}^{m\times n}$	$2mnk$
`CGS` (X)	$X \in \mathbb{R}^{m\times n}$	$2mn^2$
`CGS2` (X)	$X \in \mathbb{R}^{m\times n}$	$4mn^2$
`MGS` (X)	$X \in \mathbb{R}^{m\times n}$	$2mn^2$
`BCGS` (X)	$X \in \mathbb{R}^{m\times n}$	$2mn^2$
`BCGS2` (X)	$X \in \mathbb{R}^{m\times n}$	$4mn^2$
`BMGS` (X)	$X \in \mathbb{R}^{m\times n}$	$2mn^2$

Figure 3.6: Important flop counts

We conducted our benchmarks on two hardware configurations:

- 16 Intel Xeon Gold 6226R and NVIDIA Quadro RTX 4000
- 48 Intel(R) Xeon(R) Gold 6342 CPU and Nvidia A100 80GB PCIe

The first benchmark was conducted on a regular workstation GPU with not many double precision cores. The processing power is 222.5 GFLOPS in double precision [35]. As a result, the GPU implementation is actually slower than the CPU one. Depending on the algorithm, the CPU can be 3 to 6 times faster than the GPU.

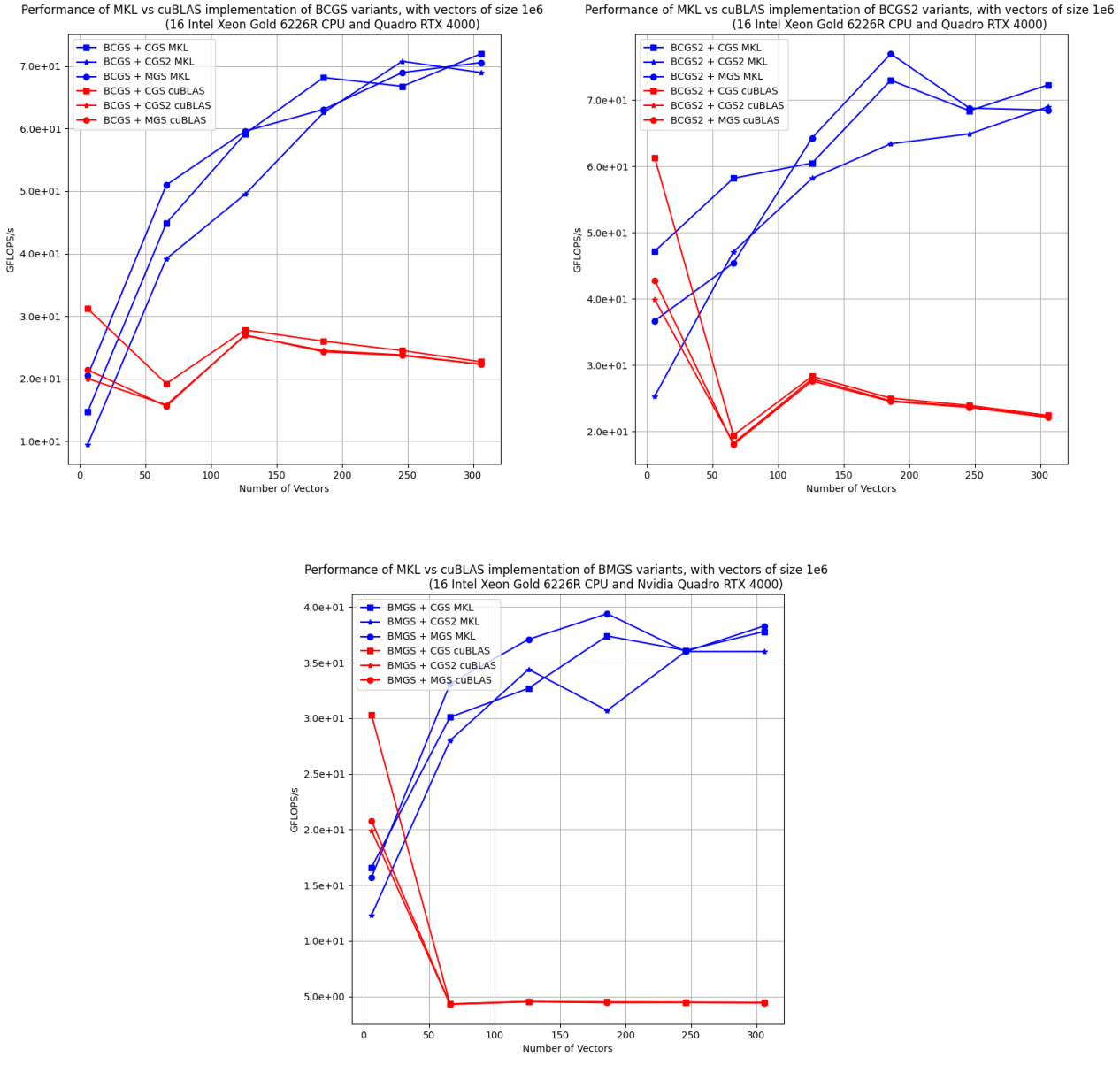

Figure 3.7: Benchmark results of CPU vs GPU on NVIDIA Quadro RTX 4000

We also conducted the same benchmark on a different GPU to see the difference between a basic workstation GPU and a professional GPU. For this model, the processing power in double precision is 9,700 GFLOPS which makes a huge difference compared to the previous one. This time, the GPU implementation achieved up to 10 times speed-up compared to the CPU as we can see in 3.8.

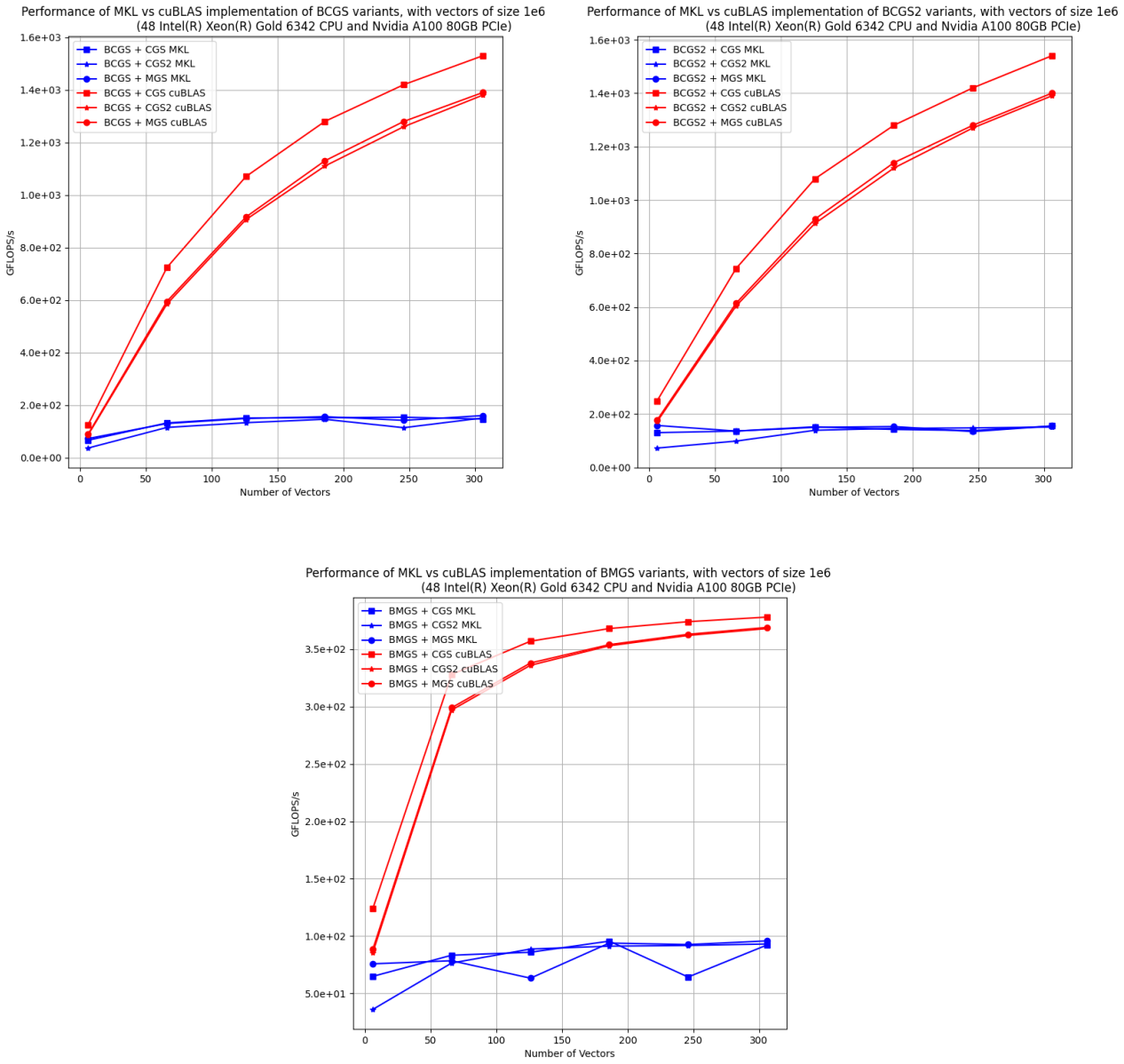

Figure 3.8: Benchmark results of CPU vs GPU on NVIDIA A100

Now that we have the results, the question we want to address is which of the tested algorithms is the best fit for our use case. For this, we compare the execution time of each BGS variants as well as their numerical properties. The goal is to find the best trade-off between performance and numerical stability.

First, we notice that the `IntraOrtho` subroutines don't have much impact on the performances of the CPU and GPU. This can be explain by the fact that the block size used for these benchmarks was $s = 6$ which is a small block size. So the difference between the various GS variants cannot significantly influence the execution time.

Then, regarding the CPU versions, the `BCGS` is faster than `BMGS` which is faster than `BCGS2`. The `BCGS` being to unstable, we only consider `BMGS` and `BCGS2`. We believe that the `BMGS` $\circ$ `CGS2` , the `BCGS2` $\circ$ `MGS` and the `BCGS2` $\circ$ `CGS2` are each a good solution. The difficulty is to find the right trade-off between numerical stability and performance that fit the use case.

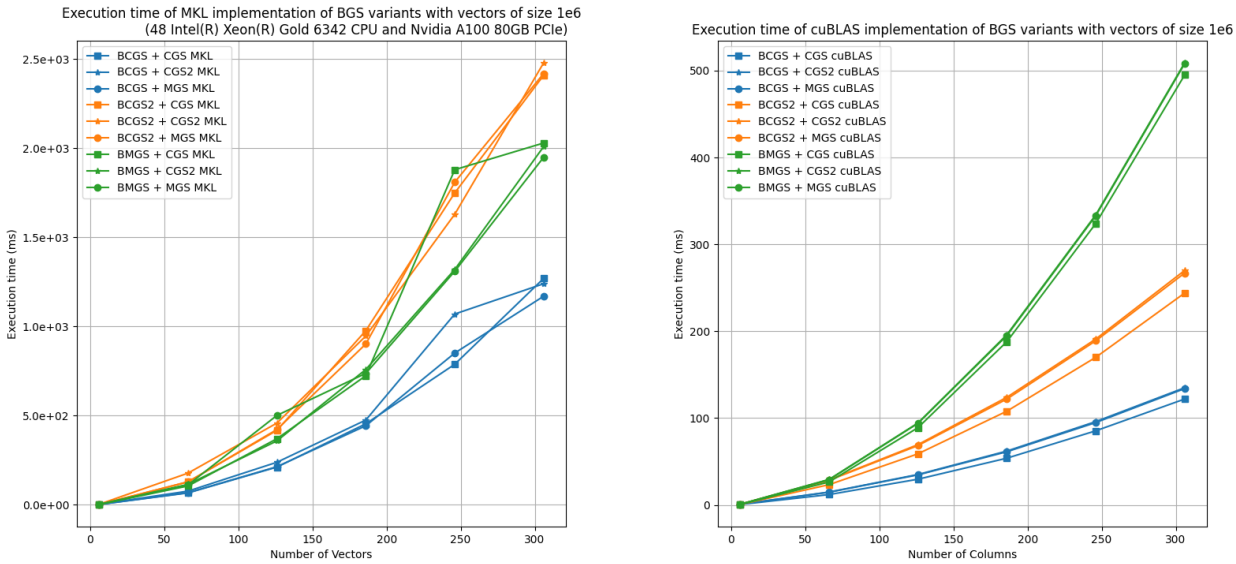

Figure 3.9: Execution time benchmark of CPU and GPU

Finally, regarding the GPU versions, the `BCGS` is faster than the `BCGS2` which is faster than the `BMGS`. Unlike the CPU version, there is no trade-off to find because the `BCGS2` is the fastest and the more stable (not considering `BCGS`). Thus, the `BCGS2` $\circ$ `CGS2` is the best choice on GPU.

To conclude this section, we managed to implement some orthogonalization algorithms that are faster on professional GPU than on CPU and so accelerate algorithms that use orthogonalization. We also demonstrated which one was the better solution regarding the performances and the numerical stability.

Unfortunately, we couldn't integrate the code into the actual Shifted Block Lanczos implementation due to architectural design constraints. Consequently, measuring the GPU acceleration of the entire algorithm was not possible. This will be addressed in future work.

3.2 Benchmark of a Direct Sparse Solver

As said in the previous section, another important computational cost is the solve of a linear system during the Shifted Block Lanczos algorithm. There is many CPU-based direct sparse solvers such as MUMPS [2] or PARDISO [32], but also GPU-accelerated ones such as CHOLMOD [9] or GLU [31]. But NVIDIA released its own direct sparse solver cuDSS [28], in November 2023, which was really promising so we decided to test it.

To test this GPU linear solver, we decided to offload the solve of the linear system to the GPU and compare the execution time of this operation with the actual CPU-based direct sparse solver of Ansys. The solve of a linear system such as $A\mathbf{x} = \mathbf{b}$ consist of 3 main phases:

1. The **symbolic factorization** is a preparatory step that focuses on understanding and optimizing the structure of the matrix factors, often involving reordering of the matrix to minimize fill-in and improve computational efficiency during the numerical factorization.
2. The **numerical factorization** decompose the matrix into a product of matrices. There are many different matrix factorizations. This step is necessary for the solve of the linear system.
3. The **solve** phase which basically solve the linear system using the factorized matrix and the right hand side of the equation.

Sometimes, we need to solve a linear system many times with the same matrix structure, but different numerical factors. In this case, we can perform the symbolic factorization only once, since it only depends on the matrix structure and not the numerical values of the factors. Then, we need to perform the numerical factorization whenever the numerical value of the matrix factors change. Finally, we can perform the solve phase for any right hand side of the linear system.

In the Shifted Block Lanczos algorithm (4), the computation involves a matrix-matrix product with the inverse of a matrix, specifically $W = (K - \sigma M)^{-1}(MQ_j)$ (see line 3). However, computing the inverse of $(K - \sigma M)$ is inefficient and sometimes impossible. Since we only require the result of the matrix-matrix product involving $(K - \sigma M)^{-1}$, we can compute it implicitly:

$$W = (K - \sigma M)^{-1}(MQ_j) \tag{3.4}$$

$$(K - \sigma M)W = (MQ_j) \tag{3.5}$$

Thus, we solve, for a given shift σ, the linear system of equations

$$(K - \sigma M)X = (MQ_j) \tag{3.6}$$

This means that we need to perform symbolic factorization only once for the whole method (even if we restart the algorithm with a different shift σ). We need also to perform numerical factorization of the matrix $(K - \sigma M)$ for each shift σ. And finally we need to solve the linear system (3.6) at each iteration.

We conducted our benchmark on different models and on different hardware configurations. The size of a model is quantified in terms of its **degrees of freedom (dof)**. For a given size of a model, the size of the matrices K and M correspond to the number of dof. The model size was constrained by the 80GB capacity of the GPU memory. The two hardware configurations were:

- `np` x Intel(R) Xeon(R) Gold 6342 CPU and NVIDIA RTX 6000 Ada Generation
- `np` x Intel(R) Xeon(R) Gold 6342 CPU and NVIDIA H100 PCIe

With variable number of cores `np` from 8 to 32.

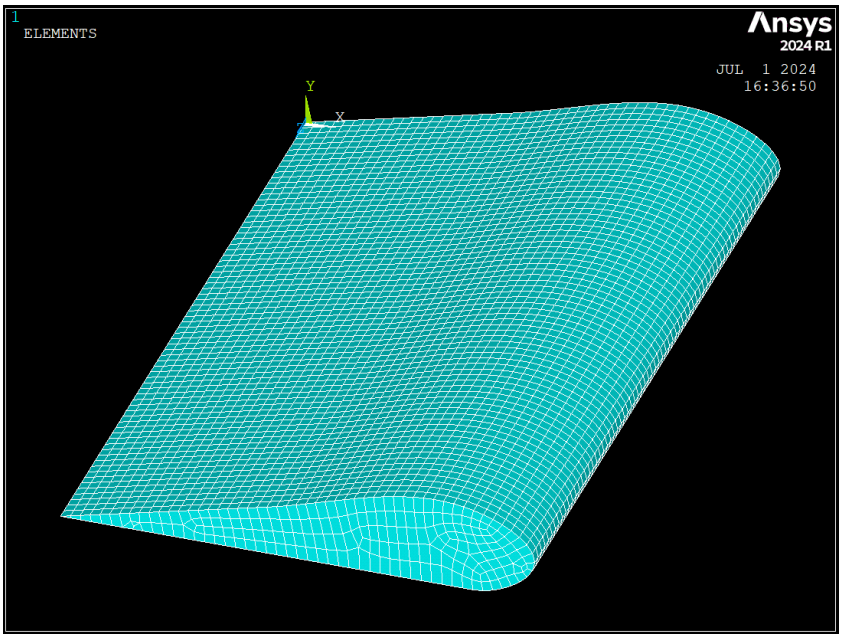

Figure 3.10: Plane wing model

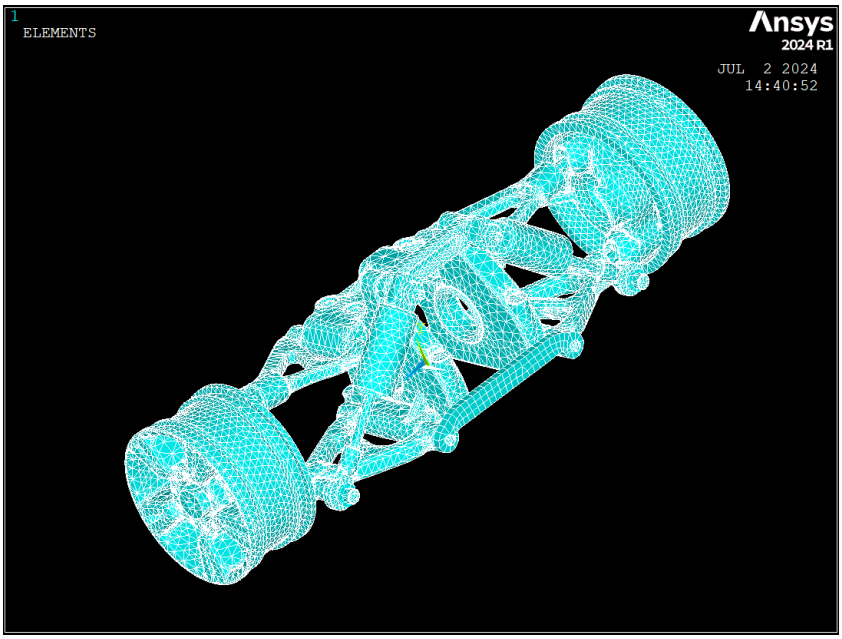

Figure 3.11: Suspension model

We performed modal analysis, searching for 50 eigenfrequencies, on a plane wing model (figure 3.10). This model was available in different sizes. Our testing covered three versions with 200,000, 1,000,000, and 2,000,000 degrees of freedom. We also used a suspension model of 1,000,000 degrees of freedom (figure 3.11).

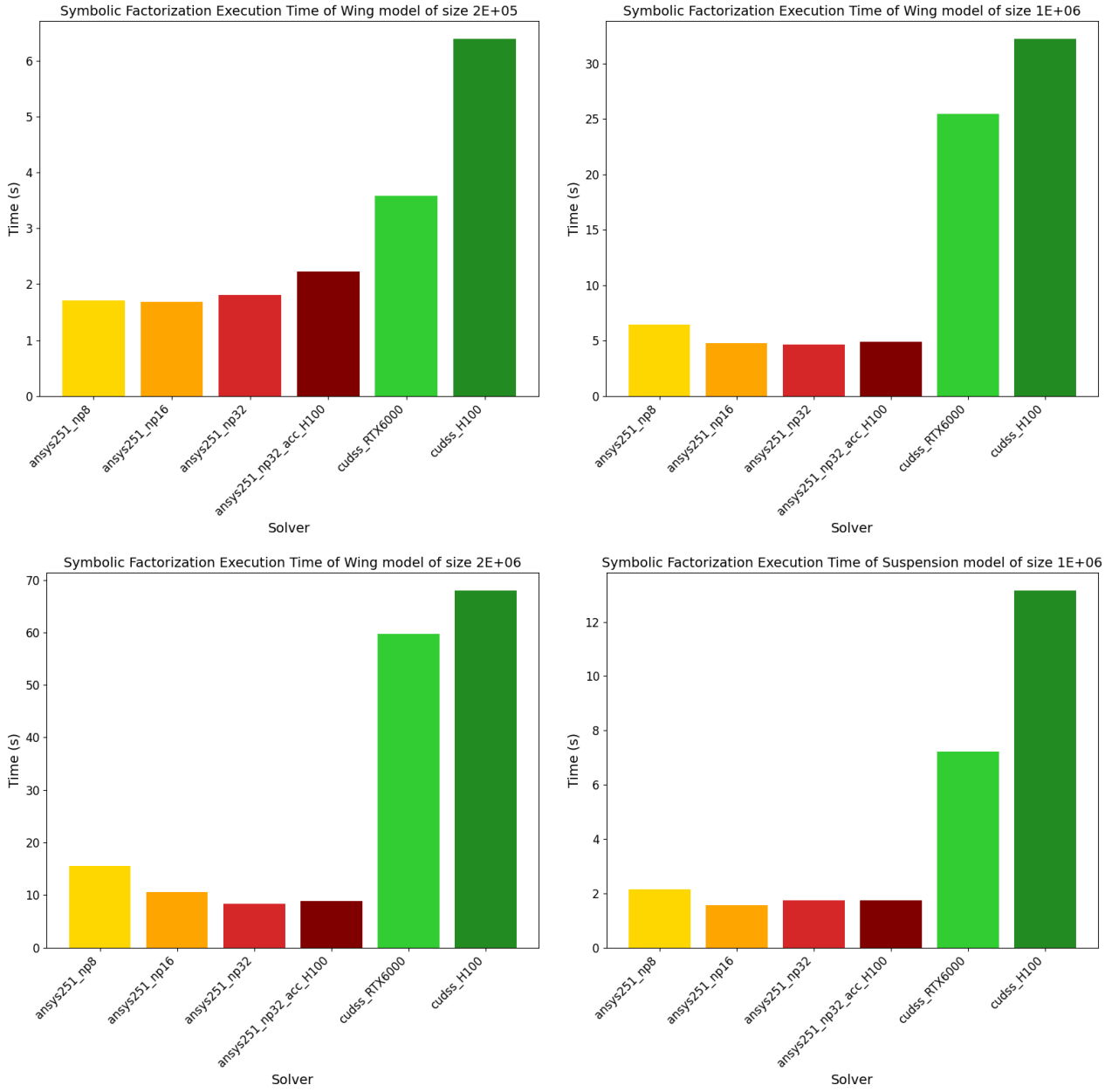

Figure 3.12: Symbolic Factorization benchmark results

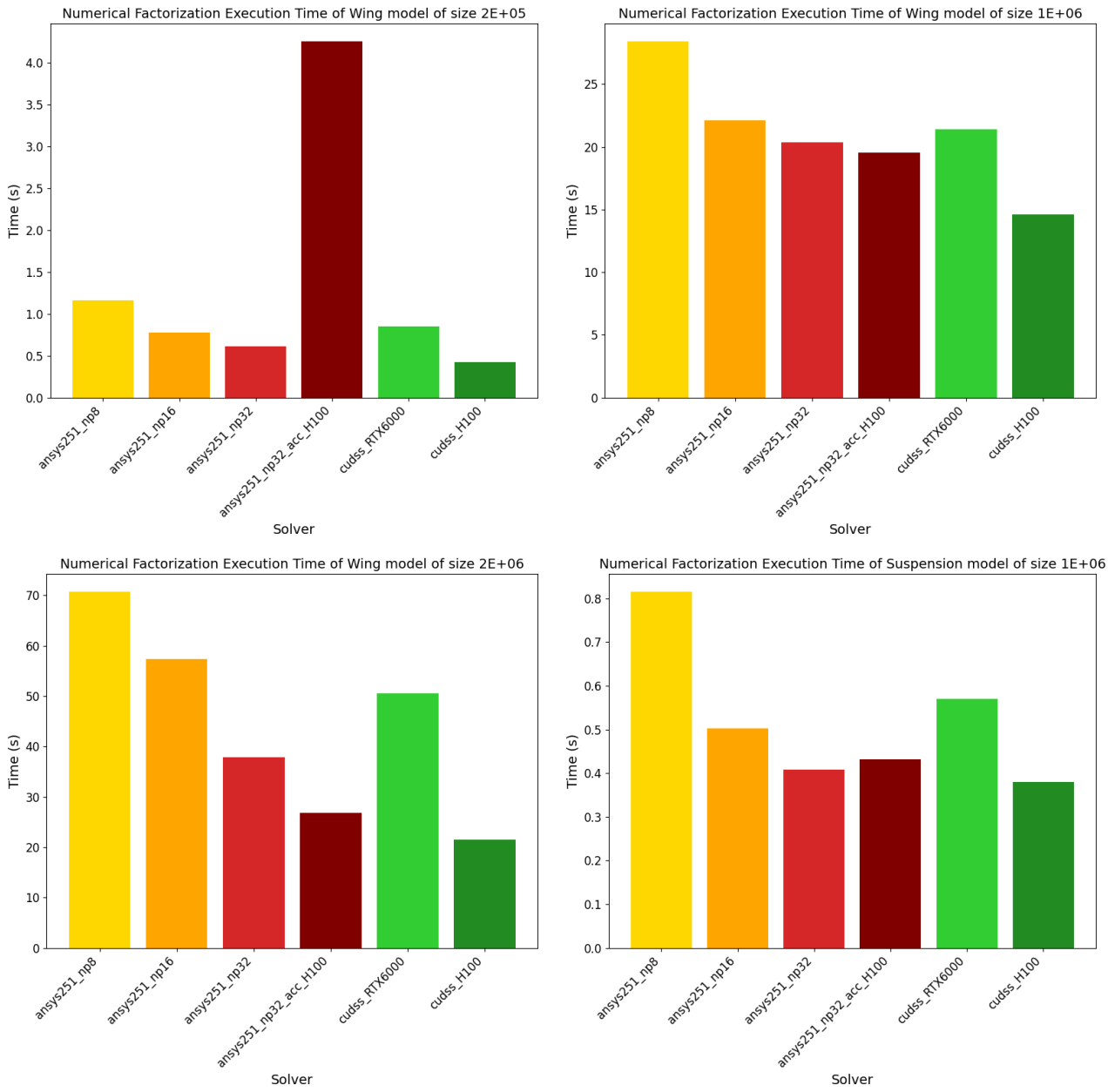

Figure 3.13: Numerical Factorization benchmark results

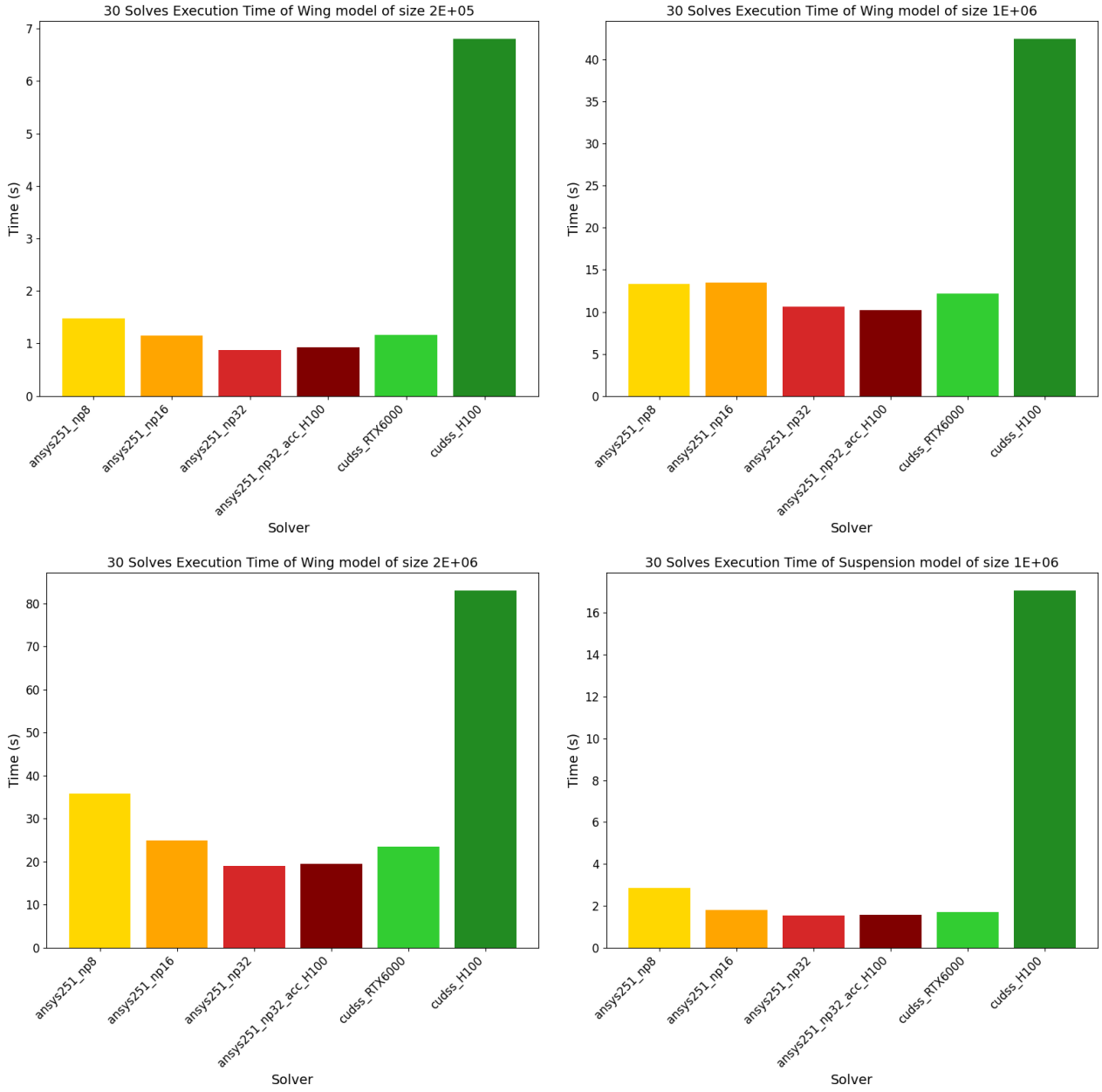

Figure 3.14: Solve benchmark results

Since we only requested 50 eigenfrequencies, no shifting was necessary during the solve process. Only one symbolic factorization and one numerical factorization were required. The algorithm needed 30 iterations to capture the 50 eigenfrequencies, resulting in the solution of 30 linear systems. The benchmark results are grouped according to the solve phase. The figure 3.12 shows the symbolic factorization results, figure 3.13 shows the numerical factorization results and figure 3.14 shows the solve results.

The labels along the x axis denote the solver used to solve the linear system. `ansys251_np8` means the Ansys's solver from the release R251 distributed on 8 cores. The `_acc_H100` term means that we enabled the GPU acceleration of the factorization phase (that is not part of our work) on a NVIDIA H100. The `cudss_RTX6000` denote the NVIDIA's cuDSS solver running on a NVIDIA RTX 6000 Ada Generation.

Firstly, we observe that both the CPU and GPU solvers scale similarly with the model size. The execution time ratios remain consistent for models of 200,000, 1,000,000, and 2,000,000 units. However, GPU acceleration for numerical factorization is only advantageous for model sizes exceeding 1,000,000.

Secondly, concerning the symbolic factorization phase, Ansys's solver is significantly faster than NVIDIA's one, being up to 6 times quicker depending on the model size. We also notice that this operation is surprisingly faster on NVIDIA RTX6000 Ada generation than on a NVIDIA H100. This outcome is unexpected because the NVIDIA H100 is currently the best GPU available on the market.

Thirdly, regarding numerical factorization, the cuDSS solver running on an NVIDIA H100 is the fastest, although it is nearly matched by the fastest of Ansys. The performance of the cuDSS solver on an NVIDIA RTX6000 Ada generation is comparable to Ansys's performance distributed across 16 cores. This indicates that GPUs significantly impact numerical factorization time, with the operation being faster on an H100 as expected.

Fourthly, regarding the solve execution time, we observe that Ansys distributed on 32 cores is the fastest one. But surprisingly, cuDSS running on a NVIDIA H100 is by far the slowest one, whereas it is closely matching the performances of Ansys distributed on 16 cores with a NVIDIA RTX6000 Ada generation.

Finally, the overall factorization and solve process is faster with Ansys's solver compared to NVIDIA cuDSS, which is good news for Ansys. However, we need to be cautious when analyzing the results, as the performance of the cuDSS solver varies significantly depending on the solve phase and the hardware used. Notably, cuDSS can even perform faster on GPUs with less number of cores.

3.3 TOAR Algorithm

As I had finished working on the initial objectives of my internship, we decided to take it a step further and work on implementing an eigensolver for Quadratic Eigenvalue Problem (QEP). The new objectives was to implement a GPU version of the Two-level Orthogonal ARnoldi (TOAR) algorithm. Let's first introduce some definitions to better understand the algorithm.

3.3.1 Second-order Krylov subspace

Let A and B be $n \times n$ matrices and $\mathbf{r}_{-1}, \mathbf{r}_0$ be length-n vectors such that $[\mathbf{r}_{-1}, \mathbf{r}_0] \neq \mathbf{0}$. Then the sequence $\mathbf{r}_{-1}, \mathbf{r}_0, \mathbf{r}_1, ...,$ with

$$\mathbf{r}_j = A\mathbf{r}_{j-1} + B\mathbf{r}_{j-2} \text{ for } j \geq 1 \tag{3.7}$$

is called a *second-order Krylov sequence* based on $A, B, \mathbf{r}_{-1}$ and $\mathbf{r}_0$. The subspace

$$\mathcal{G}_k(A, B; \mathbf{r}_{-1}, \mathbf{r}_0) = \text{span}\{\mathbf{r}_{-1}, \mathbf{r}_0, \mathbf{r}_1, ..., \mathbf{r}_{k-1}\} \tag{3.8}$$

is called a k-th *second-order Krylov subspace*.

The second-order Krylov subspace $\mathcal{G}_k(A, B; \mathbf{r}_{-1}, \mathbf{r}_0)$ can be embedded in the linear Krylov subspace

$$\mathcal{K}_k(L, \mathbf{v}_0) = \text{span } \{\mathbf{v}_0, L\mathbf{v}_0, L^2\mathbf{v}_0, ..., L^{k-1}\mathbf{v}_0\} \tag{3.9}$$

$$= \text{span } \left\{ \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{r}_{-1} \end{bmatrix}, \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_0 \end{bmatrix}, \begin{bmatrix} \mathbf{r}_2 \\ \mathbf{r}_1 \end{bmatrix}, ..., \begin{bmatrix} \mathbf{r}_{k-1} \\ \mathbf{r}_{k-2} \end{bmatrix} \right\} \tag{3.10}$$

where

$$L = \begin{bmatrix} A & B \\ I & 0 \end{bmatrix} \in \mathbb{R}^{2n \times 2n} \text{ and } \mathbf{v}_0 = \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{r}_{-1} \end{bmatrix} \in \mathbb{R}^{2n}$$

and I is an identity matrix of size n. Specifically, let $\mathcal{Q}_k$ and $\mathcal{V}_k$ be basis matrices of the subspaces $\mathcal{G}_k(A, B; \mathbf{r}_{-1}, \mathbf{r}_0)$ and $\mathcal{K}_k(L, \mathbf{v}_0)$ respectively. Then

$$\mathcal{V}_k = \begin{bmatrix} \mathcal{Q}_k \mathcal{U}_k^{[1]} \\ \mathcal{Q}_k \mathcal{U}_k^{[2]} \end{bmatrix} = \begin{bmatrix} \mathcal{Q}_k & \\ & \mathcal{Q}_k \end{bmatrix} \begin{bmatrix} \mathcal{U}_k^{[1]} \\ \mathcal{U}_k^{[2]} \end{bmatrix} = \mathcal{Q}_{[k]} \mathcal{U}_k \tag{3.11}$$

Note that $\mathcal{Q}_k$ is $n \times \eta_k$, $\mathcal{U}_k^{[1]}$ and $\mathcal{U}_k^{[2]}$ are $\eta_k \times k$, where η_k is the dimension of $\mathcal{G}_k(A, B; \mathbf{r}_{-1}, \mathbf{r}_0)$. Note also that $\eta_k \leq k + 1$ due to possible deflations. Equation (3.11) indicates a memory-efficient compact representation of the basis matrix $\mathcal{V}_k$ since the memory size $2nk$ required to store $\mathcal{V}_k$ is reduced to $(n + 2k)\eta_k$ for $\mathcal{Q}_k$ and $\mathcal{U}_k$.

3.3.2 TOAR algorithm

The TOAR algorithm (13) compute an orthogonal basis matrix $\mathcal{Q}_k$ of the second-order Krylov subspace $\mathcal{G}_k(A, B; \mathbf{r}_{-1}, \mathbf{r}_0)$ while maintaining the basis matrix $\mathcal{V}_k$ of the associated linear Krylov subspace $\mathcal{K}_k(L, \mathbf{v}_0)$ as orthogonal.

Recall that the Arnoldi procedure computes an orthogonal basis of the k-th Krylov subspace $\mathcal{K}_k(L, \mathbf{v}_0)$ such that

$$\mathcal{H}_k = \mathcal{V}_k^T L \mathcal{V}_k \tag{3.12}$$

see equation (2.13) for more details. From equation (3.11) we have

$$\mathcal{H}_k = \begin{bmatrix} \mathcal{Q}_k \mathcal{U}_k^{[1]} \\ \mathcal{Q}_k \mathcal{U}_k^{[2]} \end{bmatrix}^T \begin{bmatrix} A & B \\ I & 0 \end{bmatrix} \begin{bmatrix} \mathcal{Q}_k \mathcal{U}_k^{[1]} \\ \mathcal{Q}_k \mathcal{U}_k^{[2]} \end{bmatrix} \tag{3.13}$$

The TOAR algorithm computes $\mathcal{Q}_k$, $\mathcal{U}_k^{[1]}$ and $\mathcal{U}_k^{[2]}$ without explicitly generating $\mathcal{V}_k$.

3.3.3 Implementation

Algorithm 13 Two-level Orthogonal ARnoldi (TOAR) Algorithm

Input: $A, B \in \mathbb{R}^{n \times n}$ and $r_{-1}, r_0 \in \mathbb{R}^n$ with $\gamma = \|[r_{-1}, r_0]\|_F \neq 0$ and subspace order k
Output: $\mathcal{Q}_k \in \mathbb{R}^{n \times \eta_k}, \mathcal{U}_{k,1}, \mathcal{U}_{k,2} \in \mathbb{R}^{\eta_k \times k}$ and $\mathcal{H}_k = \{h_{ij}\} \in \mathbb{R}^{k \times k-1}$

$QX = \texttt{RRQR}([r_{-1}, r_0])$ with η_1 being the rank
Initialize $\mathcal{Q}_1 = Q, \mathbf{u}_1^{[1]} = X_{:,2}/\gamma$ and $\mathbf{u}_1^{[2]} = X_{:,1}/\gamma$
for $j = 1, 2, ..., k-1$ **do**
  $\mathbf{r} = A(\mathcal{Q}_j \mathbf{u}_j^{[1]}) + B(\mathcal{Q}_j \mathbf{u}_j^{[2]})$
  for $i = 1, ..., \eta_j$ **do** `# Modified Gram-Schmidt`
    $s_i = \mathbf{q}_i^T \mathbf{r}$
    $\mathbf{r} = \mathbf{r} - s_i \mathbf{q}_i$
  end for
  $\alpha = \|\mathbf{r}\|_2$
  Set $\mathbf{s} = [s_1, ..., s_{\eta_j}]^T$
  for $i = 1, ..., j$ **do** `# Modified Gram-Schmidt`
    $h_{ij} = \mathbf{u}_i^{[1]T} \mathbf{s} + \mathbf{u}_i^{[2]T} \mathbf{u}_j^{[1]}$
    $\mathbf{s} = \mathbf{s} - h_{ij} \mathbf{u}_i^{[1]}$
    $\mathbf{u}_j^{[1]} = \mathbf{u}_j^{[1]} - h_{ij} \mathbf{u}_i^{[2]}$
  end for
  $h_{j+1,j} = (\alpha^2 + \|\mathbf{s}\|_2^2 + \|\mathbf{u}\|_2^2)^{1/2}$
  if $h_{j+1,j} = 0$ **then** `# breakdown`
    stop
  end if
  if $\alpha = 0$ **then** `# deflation`
    $\mathcal{U}_{j+1}^{[1]} = \begin{bmatrix} \mathcal{U}_j^{[1]} & s/h_{j+1,j} \end{bmatrix} \quad \mathcal{U}_{j+1}^{[2]} = \begin{bmatrix} \mathcal{U}_j^{[2]} & \mathbf{u}_j^{[1]}/h_{j+1,j} \end{bmatrix}$
  else
    $\eta_{j+1} = \eta_j + 1$
    $\mathcal{Q}_{j+1} = \begin{bmatrix} \mathcal{Q}_j & \mathbf{r}/\alpha \end{bmatrix}$
    $\mathcal{U}_{j+1}^{[1]} = \begin{bmatrix} \mathcal{U}_j^{[1]} & \mathbf{s}/h_{j+1,j} \\ 0 & \alpha/h_{j+1,j} \end{bmatrix} \quad \mathcal{U}_{j+1}^{[2]} = \begin{bmatrix} \mathcal{U}_j^{[2]} & \mathbf{u}_j^{[1]}/h_{j+1,j} \\ 0 & 0 \end{bmatrix}$
  end if
end for

We see that we already have most of the tools to implement this algorithm. Firstly, this algorithm require some basic linear algebra operations most of the time provided by the BLAS libraries.

Secondly, we already studied the Gram-Schmidt orthogonalization that is present line 5 and 11. We then rely on our work concerning orthogonalization procedure (see subsection 3.1). We will also use the fact that for a relatively big problem size, the `CGS2` is actually faster than `MGS` on GPU.

Thirdly, The `RRQR()` subroutines is the Rank Revealing QR which is basically a QR factorization that also return the rank of the resulting matrix R. This operation can easily be implemented with Gram-Schmidt algorithm.

Finally, at line 4, we solve a linear system in order to implicitly compute the result of the matrix-vector product with A and B since these matrices are implicitly given by $A = -K^{-1}C$ and $B = -K^{-1}M$ (see equation (2.20)). We will use the cuDSS linear solver that we tested previously (see section 3.2).

3.3.4 Shifting the QEP

As we did with the generalized eigenvalue problem (2.8), we can shift the the quadratic eigenvalue problem. Therefore, instead of computing $A = -K^{-1}C$ and $B = -K^{-1}M$, we compute $A = -(\sigma^2 M + \sigma C + K)^{-1}M$ and $B = -(\sigma^2 M + \sigma C + K)^{-1}C$ for a given shift σ.

I did not have sufficient time to fully implement the method described in (2.8). We managed to complete only the first step, which involves computing the reduced subspace using the TOAR algorithm. Evaluating the performance of the algorithm is not straightforward because we are unable to compute the final eigenvalues. However, since the TOAR algorithm is supposed to generate an orthogonal subspace $\mathcal{Q}_k$, we can at least measure its orthogonality. The following table presents the results for various problems:

Problem size	Subspace size η	$\|I_\eta - \hat{\mathcal{Q}}_\eta^T \hat{\mathcal{Q}}_\eta\|$
$M, C, K \in \mathbb{R}^{10\times 10}$	10	6.2e-16
$M, C, K \in \mathbb{R}^{639\times 639}$	455	1.5e-14
$M, C, K \in \mathbb{R}^{915\times 915}$	7	5.1e-16
$M, C, K \in \mathbb{R}^{1940\times 1940}$	201	9.7e-15

Figure 3.15: Results of TOAR algorithm

We can see that the orthogonality of $\mathcal{Q}_\eta$ is well preserved. However, we notice that the subspace sizes are much smaller than the original matrices size. This is due to deflation in the TOAR algorithm. This need to be addressed in a future work because the size of the subspace may be not big enough the capture all the desired eigenvalues.

Chapter 4

Conclusion

In conclusion, the initial goal of accelerating certain parts of an eigensolver has been achieved. Firstly, we were able to identify a better orthogonalization algorithm on the GPU. The advantages of the GPU enable a faster and more numerically robust procedure. However, this result depends on the GPU used. Next, we tested a direct sparse solver, which proved to be interesting as it is faster for numerical factorization phase when solving the linear system. However, it ultimately turned out to be slower and less numerically robust than Ansys's current solver. Finally, we started developing a new GPU-based eigensolver for quadratic eigenvalue problems. Although not yet fully implemented, it's a good start and will be the subject of future work.

Overall, the work carried out is promising and has generated significant interest among several other teams. The work will be continued in the future. I've met some brilliant people who are experts in their field, and worked on some exciting, cutting-edge topics. I am very proud of the results I achieved and I am very pleased to have explored the field of numerical methods and high-performance computing. This internship has been very enriching for me, both technically and personally.

Bibliography

[1] Nabih N. Abdelmalek. "Round off error analysis for Gram-Schmidt method and solution of linear least squares problems". eng. In: *BIT Numerical Mathematics* 11.4 (1971). Record identifier / Identificateur de l'enregistrement : 7f6a690a-2cba-47c6-96f5-e1cfbf6309cd, pages 345–367. ISSN: 1572-9125. DOI: 10.1007/BF01939404. Accessible on: https://doi.org/10.1007/BF01939404.

[2] Patrick R. Amestoy et al. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM J. Matrix Anal. Appl.* 23 (2001), pages 15–41. Accessible on: https://api.semanticscholar.org/CorpusID:2329523.

[3] E. Anderson et al. "LAPACK: a portable linear algebra library for high-performance computers". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pages 2–11. ISBN: 0897914120.

[4] Walter E. Arnoldi. "The principle of minimized iterations in the solution of the matrix eigenvalue problem". In: *Quarterly of Applied Mathematics* 9 (1951), pages 17–29. Accessible on: https://api.semanticscholar.org/CorpusID:115852469.

[5] Jesse L. Barlow and Alicja Smoktunowicz. "Reorthogonalized block classical Gram-Schmidt". English (US). In: *Numerische Mathematik* 123.3 (Mar. 2013). Funding Information: The research of J. L. Barlow was sponsored by the National Science Foundation under contract no. CCF-1115704., pages 395–423. ISSN: 0029-599X. DOI: 10.1007/s00211-012-0496-2.

[6] Åke Björck. "Solving linear least squares problems by Gram-Schmidt orthogonalization". In: *BIT Numerical Mathematics* 7 (1967), pages 1–21. Accessible on: https://api.semanticscholar.org/CorpusID:119767729.

[7] S. Blackford and J. Dongarra. LAPACK Working note 41. 1999, page 118. Accessible on: https://www.netlib.org/lapack/lawnspdf/lawn41.pdf.

[8] Erin Carson et al. "Block Gram-Schmidt algorithms and their stability properties". In: *Linear Algebra and its Applications* 638 (2022), pages 150–195. ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2021.12.017. Accessible on: https://www.sciencedirect.com/science/article/pii/S0024379521004523.

[9] Yanqing Chen et al. "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate". In: *ACM Trans. Math. Softw.* 35.3 (2008). ISSN: 0098-3500. DOI: 10.1145/1391989.1391995. Accessible on: https://doi.org/10.1145/1391989.1391995.

[10] J. Dongarra and F. Sullivan. "Guest Editors Introduction to the top 10 algorithms". In: *Computing in Science & Engineering* 2.1 (2000), pages 22–23. DOI: 10.1109/MCISE.2000.814652.

[11] J. G. F. Francis. "The QR Transformation A Unitary Analogue to the LR Transformation—Part 1". In: *The Computer Journal* 4.3 (Jan. 1961), pages 265–271. ISSN: 0010-4620. DOI: 10.1093/comjnl/4.3.265. eprint: https://academic.oup.com/comjnl/article-pdf/4/3/265/1080833/040265.pdf.

[12] Luc Giraud et al. "Rounding error analysis of the classical Gram-Schmidt orthogonalization process". In: *Numerische Mathematik* 101 (2005), pages 87–100. Accessible on: https://api.semanticscholar.org/CorpusID:6900273.

[13] Gene H. Golub and Charles F. Van Loan. Matrix Computations. Third. The Johns Hopkins University Press, 1996.

[14] Roger G. Grimes, John G. Lewis, and Horst D. Simon. "A Shifted Block Lanczos Algorithm for Solving Sparse Symmetric Generalized Eigenproblems". In: *SIAM Journal on Matrix Analysis and Applications* 15.1 (1994), pages 228–272. DOI: 10.1137/S0895479888151111.

[15] Alston S. Householder. "Unitary Triangularization of a Nonsymmetric Matrix". In: *J. ACM* 5.4 (1958), pages 339–342. ISSN: 0004-5411. DOI: 10.1145/320941.320947. Accessible on: https://doi.org/10.1145/320941.320947.

[16] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pages 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[17] Intel. Intel oneAPI Math Kernel Library. Accessible on: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[18] W. Jalby and B. Philippe. "Stability Analysis and Improvement of the Block Gram–Schmidt Algorithm". In: *SIAM Journal on Scientific and Statistical Computing* 12.5 (1991), pages 1058–1073. DOI: 10.1137/0912056. Accessible on: https://doi.org/10.1137/0912056.

[19] A Kiełbasiński. "Analiza numeryczna algorytmu ortogonalizacji Grama-Schmidta". In: *Journals of the Polish Mathematical Society* 2.2 (1974), pages 15–35. DOI: 10.14708/ma.v2i2.1048. Accessible on: https://wydawnictwa.ptm.org.pl/index.php/matematyka-stosowana/article/viewArticle/1048.

[20] A. N. Krylov. "On the numerical solution of equations whose solution determine the frequency of small vibrations of material systems". In: *SSSR Otd Mat. Estest* 1 (1931), pages 491–539.

[21] Cornelius Lanczos. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators". In: *J. Res. Natl. Bur. Stand. B* 45 (1950), pages 255–282. DOI: 10.6028/jres.045.026.

[22] C. L. Lawson et al. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (1979), pages 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. Accessible on: https://doi.org/10.1145/355841.355847.

[23] Feng Li et al. "CPU versus GPU: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms". In: *Neural Computing and Applications* 31 (Aug. 2019). DOI: 10.1007/s00521-018-3354-z.

[24] Ding Lu, Yangfeng Su, and Zhaojun Bai. "Stability Analysis of the Two-level Orthogonal Arnoldi Procedure". In: *SIAM Journal on Matrix Analysis and Applications* 37.1 (2016), pages 195–214. DOI: 10.1137/151005142.

[25] R. V. Mises and H. Pollaczek-Geiringer. "Praktische Verfahren der Gleichungsauflösung ." In: *Zeitschrift Angewandte Mathematik und Mechanik* 9.1 (Jan. 1929), pages 58–77. DOI: 10.1002/zamm.19290090105.

[26] NVIDIA. cuBLAS. Accessible on: https://docs.nvidia.com/cuda/cublas/.

[27] NVIDIA. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. 2024.

[28] NVIDIA. cuDSS. Accessible on: https://developer.nvidia.com/cudss.

[29] NVIDIA. cuSPARSE. Accessible on: https://docs.nvidia.com/cuda/cusparse/index.html.

[30] Beresford N. Parlett. The Symmetric Eigenvalue Problem. Society for Industrial and Applied Mathematics, 1998. DOI: 10.1137/1.9781611971163. Accessible on: https://epubs.siam.org/doi/abs/10.1137/1.9781611971163.

[31] Shaoyi Peng and Sheldon X.-D. Tan. "GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation". In: *CoRR* abs/1908.00204 (2019). arXiv: 1908.00204. Accessible on: http://arxiv.org/abs/1908.00204.

[32] Olaf Schenk, Klaus Gärtner, and Wolfgang Fichtner. "Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors". In: *BIT Numerical Mathematics* 40 (2000), pages 158–176. Accessible on: https://api.semanticscholar.org/CorpusID:59634216.

[33] Françoise Tisseur and Karl Meerbergen. "The Quadratic Eigenvalue Problem". In: *SIAM Review* 43.2 (2001), pages 235–286. DOI: 10.1137/S0036144500381988.

[34] Wikipedi. A sparse matrix obtained when solving a finite element problem in two dimensions. The non-zero elements are shown in black. Accessible on: https://en.wikipedia.org/wiki/Sparse_matrix.

[35] Wikipedia. List of Nvidia graphics processing units. Accessible on: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units.

[36] Yusaku Yamamoto et al. "Roundoff error analysis of the Cholesky QR2 algorithm". English. In: *Electronic Transactions on Numerical Analysis* 44 (2015), pages 306–326. ISSN: 1068-9613.

Gabriel Schwab

Internship Report ST50 - P2024

Internship subject and description of the work done

Solvers are computational tools or algorithm used to find numerical solution of equation or system of equations. In particular, eigensolver finds eigenvalues and eigenvectors of a matrix, which are crucial in many scientific and engineering applications.

Performance is a very important criterion for the development of these eigensolvers, which are required to solve very large problems. GPUs are hardware specially designed to accelerate linear algebra operations through parallel computing. So it's worth taking advantage of these capabilities to speed up solver computations.

During this internship, I identified some orthogonalization algorithms that are more efficient on the GPU than on the CPU, making it possible to speed up certain phases of the solving process. I also benchmarked a new linear solver, and compared it to Ansys's linear solver. Finally, I started the GPU-based implementation of a new solver for quadratic eigenvalue problems.

Ansys France
35-37 Rue Louis Guérin,
69100 Villeurbanne
https://www.ansys.com