Performance Analysis of the Incomplete Cholesky Preconditioned Conjugate Gradient Method on NVIDIA Graphics Processing Units with MATLAB

Vivian Yi Fen Yong,

School of Electrical & Computer Engineering

McGill University, Montreal

December 2023

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Masters of Electrical Engineering

Abstract

In an era where solving intricate linear systems is a commonplace task across various domains, the need for computational efficiency remains paramount. This thesis seeks to bridge the gap between complex mathematical algorithms and accessibility for engineers, researchers, scientists, and enthusiasts alike.

At its core, this research delves into the synergies between two contemporary computational technologies: the Incomplete Cholesky Preconditioned Conjugate Gradient (ICPCG) method and modern Graphics Processing Units (GPUs), with a particular focus on NVIDIA mobile graphics chips. The ICPCG method is renowned for its effectiveness in tackling large sparse systems of linear equations. However, rather than diving into the intricacies of GPU architecture with the use of an Application Programming Interface (API), such as Compute Unified Device Architecture (CUDA), we look at higher-level programming that is a more user-friendly avenue.

The ICPCG method is implemented in the MATLAB environment and utilizes the Parallel Computing Toolbox (PCT) to parallelize the method on modern NVIDIA mobile GPUs. With the use of PCT, instead of CUDA, it removes the formidable barrier of requiring an in-depth understanding of GPU hardware, often a daunting obstacle for the uninitiated. By democratizing GPU parallelization, we empower individuals from various backgrounds to harness the remarkable computational capabilities of modern GPUs without being burdened by the complexities of CUDA programming.

Chapters elucidate the ICPCG method, introduce GPU advantages over Central Processing Units (CPUs), and showcase MATLAB PCT's accessibility. A detailed methodology for implementing ICPCG on NVIDIA GPUs is provided, and the experimental results are presented in a comprehensible manner. In-depth discussions and conclusions bring forth the significance of this approach in the realm of scientific computing.

As we navigate the nexus of mathematical sophistication and accessibility, this research illuminates a path for individuals to leverage GPU parallelization effectively, transcending the boundaries of traditional CPU-based computations. In doing so, it empowers a diverse spectrum

of users to tap into the extraordinary potential of GPU-accelerated computing without the need for an advanced understanding of GPU hardware intricacies, ultimately democratizing high-performance scientific computing. Our results have showcased the benefits of parallelizing the algorithm on NVIDIA mobile GPUs, particularly for single-precision data types, while acknowledging limitations in the case of double-precision data types.

Abrégé

À une époque où la résolution de systèmes linéaires complexes est une tâche courante dans divers domaines, le besoin d'efficacité informatique reste primordial. Cette thèse cherche à combler le fossé entre les algorithmes mathématiques complexes et l'accessibilité pour les ingénieurs, les chercheurs, les scientifiques et les passionnés.

À la base, cette recherche explore les synergies entre deux technologies informatiques contemporaines: la méthode Incomplete Cholesky Preconditioned Conjugate Gradient (ICPCG) et les unités de traitement graphique (en anglais, Graphics Processing Units, ou GPUs) modernes, avec un accent particulier sur les puces graphiques mobiles NVIDIA. La méthode ICPCG est réputée pour son efficacité dans le traitement de grands systèmes clairsemés d'équations linéaires. Cependant, plutôt que de plonger dans les subtilités de l'architecture GPU avec l'utilisation d'une interface de programmation d'application (en anglais, Application Programming Interface, ou API), telle que Compute Unified Device Architecture (CUDA), nous examinons une programmation de niveau supérieur qui constitue une voie plus conviviale.

La méthode ICPCG est implémentée dans l'environnement MATLAB et utilise Parallel Computing Toolbox (PCT) pour paralléliser la méthode sur les GPU NVIDIA modernes. Avec l'utilisation de PCT, au lieu de CUDA, il supprime la formidable barrière consistant à exiger une compréhension approfondie du matériel GPU, souvent un obstacle de taille pour les non-initiés. En démocratisant la parallélisation des GPU, nous permettons à des individus d'horizons divers d'exploiter les remarquables capacités de calcul des GPU modernes sans être gênés par les complexités de la programmation CUDA.

Les chapitres expliquent la méthode ICPCG, présentent les avantages du GPU par rapport aux unités centrales de traitement (en anglais, Central Processing Unit, ou CPU) et présentent l'accessibilité du MATLAB PCT. Une méthodologie détaillée pour implémenter ICPCG sur les GPU NVIDIA est fournie et les résultats expérimentaux sont présentés de manière compréhensible. Des discussions et des conclusions approfondies font ressortir l'importance de cette approche dans le domaine du calcul scientifique.

Alors que nous naviguons entre la sophistication mathématique et l'accessibilité, cette recherche ouvre la voie aux individus pour exploiter efficacement la parallélisation GPU, transcendant les limites des calculs traditionnels basés sur CPU. Ce faisant, il permet à un large éventail d'utilisateurs d'exploiter le potentiel extraordinaire du calcul accéléré par GPU sans avoir besoin d'une compréhension avancée des subtilités du matériel GPU, démocratisant ainsi le calcul scientifique haute performance. Nos résultats ont montré les avantages de la parallélisation de l'algorithme sur les GPU mobiles NVIDIA, en particulier pour les types de données simple précision, tout en reconnaissant les limites dans le cas des types de données double précision.

Acknowledgements

Embarking on my master's degree during the pandemic brought forth a lot of uncertainties and profound isolation. These trying times, compounded with other challenges, presented a considerable amount of stress. Nevertheless, I am immensely grateful for the unwavering support of those who stood by my side, including my little Shih Tzu, Yeti, who has been by my side, providing emotional comfort throughout this journey.

Foremost, I extend my heartfelt gratitude to my thesis supervisor, Professor Dennis Giannacopoulos. His invaluable wisdom and knowledge were instrumental in guiding me throughout this journey. Professor Dennis Giannacopoulos' patience, kindness, and reassurance instilled in me the confidence to move on this path, and for that, I am profoundly thankful.

I deeply cherish Miguel for his ceaseless support. As both a schoolmate and a cherished friend, we have weathered the storms of challenging times together, offering each other unwavering support and guidance.

Lastly, I can never adequately express my appreciation to my mother, Lisa, for her unceasing support in every conceivable way. Her unwavering assistance has been nothing short of invaluable. Equally, I am extremely thankful to Valerie and Vanessa for their unwavering emotional support, their constant displays of affection, and for standing beside me through all ups and downs. With them by my side, I found the strength to navigate through the arduous moments and remain resilient.

Contribution of Authors

This section serves to assert that the work contained within this thesis was conducted and executed by the author, Yong Yi Fen Vivian. The author has undertaken the task of implementing the ICPCG method within MATLAB, employing the PCT to facilitate parallelization. Furthermore, it is worth acknowledging that the foundational code, other than the ICPCG method, has its roots in MATLAB.

Table of Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
CG	Conjugate Gradient
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFFT	Discrete Fast Fourier Transform
DRAM	Dynamic Random-Access Memory
ECC	Error-Correcting Code
FFT	Fast Fourier Transform
FLOPS	Floating-Point Operations Per Second
GB	Gigabytes
GFLOPS	Gigaflops
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
IC	Incomplete Cholesky
ICP	Incomplete Cholesky Preconditioner
ICPCG	Incomplete Cholesky Preconditioner Conjugate Gradient
LU	Lower-Upper
MB	Megabytes
PCG	Preconditioned Conjugate Gradient

PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCT	Parallel Computing Toolbox
PD	Positive Definite
RAM	Random-Access Memory
RHS	Right-Hand Side
RO	Read-Only
SIMD	Single Instruction Multiple Data
SIMT	Single-Instruction, Multiple Thread
SM	Streaming Multiprocessor
SPD	Symmetric Positive Definite
SPMD	Single Program Multiple Data
SVD	Singular Value Decomposition
TBC	Thread Block Cluster
TDP	Thermal Design Power
TPCG	Transformed Preconditioned Conjugate Gradient
UPCG	Untransformed Preconditioned Conjugate Gradient

Chapter 1
Introduction

1.1 Motivation

In the realm of scientific and engineering computing, the efficient solution of large sparse linear systems plays a pivotal role across a multitude of disciplines, from computational physics and computer graphics to data analysis and machine learning. These systems often underpin complex simulations, optimizations, and numerical modelling tasks that are essential for advancing our understanding of engineering design. As the scale and complexity of these problems continue to grow, so does the demand for innovative solutions and computing platforms that are capable of meeting these computational challenges.

GPUs, including mobile graphics chips, have emerged as formidable computational accelerators for a wide range of scientific and numerical applications. Unlike a traditional CPU, which consists of no more than a handful of cores, a GPU has a massively parallel array of integer, floating-point processors, and a dedicated high-speed memory. Typically, a GPU contains hundreds or even thousands of smaller processors. Figure 1.1 shows an example of the number of cores on a CPU and a GPU [1]. Due to their massively parallel architecture, GPUs, which were initially designed to accelerate graphics rendering, have been increasingly applied to perform general-purpose computations. As GPUs excel at parallelism, they make a particularly well-suited platform for accelerating iterative solvers commonly used to tackle large sparse linear systems [2]. Among these solvers, the preconditioned conjugate gradient (PCG) method stands out as a powerful iterative algorithm [3].

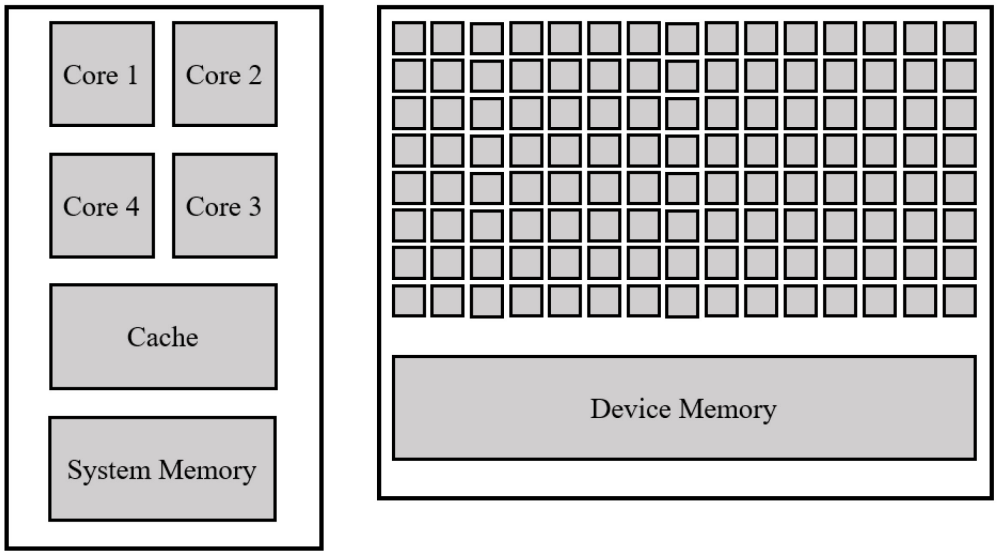

Figure 1.1: Comparison of the number of cores on a CPU system and a GPU. Left shows a CPU with multiple cores, and right shows a GPU with hundreds of cores.

Preconditioning techniques, which aim to transform the original linear system into an equivalent one with improved numerical properties, are often used in conjunction with conjugate gradient (CG), which results in PCG, to enhance its convergence speed and robustness in a wide range of applications. Preconditioning can also significantly reduce the number of iterations required for convergence [4]. In this context, the focus is on the incomplete Cholesky preconditioner (ICP). The ICP is a fundamental tool in solving large systems of linear equations as it leverages the inherent sparsity structure of the matrix to approximate the Cholesky factorization, which effectively mitigates the ill-conditioning of the system.

This thesis explores the synergies between two contemporary computational technologies: the ICPCG method and modern GPUs, with a specific emphasis on NVIDIA mobile GPUs. The ICPCG method will be implemented within MATLAB and will utilize PCT for parallelization. This toolbox does not require the use of an API, such as CUDA for NVIDIA GPUs [5]. Although CUDA has a generic parallel programming model in a multithreaded environment, it requires programmers to have a good understanding of the CUDA-supported GPU devices' hardware to fully optimize their performance. It also requires a good low-level programming skill. Otherwise, performance can vary greatly [6]. Programmers who work on languages that are not supported by CUDA can find it difficult and time-consuming to acquire the skill to implement CUDA correctly.

Hence, the primary aim of this study is to evaluate the effectiveness of using MATLAB PCT to implement ICPCG when executed on state-of-the-art NVDIA mobile GPU. The following key aspects will be examined during the research:

1. Performance of GPU Parallelism: GPUs are designed for parallelism, but to harness their power effectively for iterative solvers requires a profound understanding of their architecture and programming models. However, with MATLAB PCT, it is possible to parallelize the ICPCG algorithm without the profound understanding. Hence, this research investigates strategies to efficiently parallelize the algorithm with PCT, so that it exploits the full computational capabilities of modern GPUs [5].

2. ICPCG Method: An in-depth evaluation of incomplete Cholesky (IC) preconditioning strategies on the CG method tailored for GPU acceleration. This includes an assessment of the computational cost within the context of ICPCG.

3. Scalability of Problems: A meticulous analysis of the scalability of ICPCG on GPUs concerning problem size, sparsity pattern, and GPU hardware configuration. This study helps to determine the practical limitations and benefits of employing GPUs by using MATLAB PCT on NVIDIA mobile GPUs for solving large sparse linear systems effectively.

4. Real-world Applications: By demonstrating the performance of ICPCG on GPU using MATLAB PCT in scientific and engineering domains, it will serve as tangible demonstrations of the method's potential to expedite simulations and enhance the efficiency of solving critical, large-scale computational problems.

5. Software and Tools: Discussion of the software environment, MATLAB, and its essential toolboxes, including PCT, that facilitate the implementation of ICPCG on GPUs. This demonstrates how accessible and user-friendly MATLAB PCT is for those not well-versed in CUDA.

This research endeavours to unlock the potential of combining ICPCG on MATLAB with modern NVIDIA GPUs, enabling researchers and engineers to efficiently address complex, large-scale computational problems. The insights gained through this investigation will contribute to the

optimization of numerical simulations, ultimately enhancing our ability to tackle increasingly intricate challenges in science and engineering.

1.2 Thesis Structure

This thesis is organized into six chapters. Chapter 2 delivers an extensive review of the CG method, including the background knowledge and related work. Chapter 3 presents the key advantages of GPUs over CPUs, along with an exploration of GPU architecture to achieve these key advantages. This chapter also presents MATLAB PCT that is used for GPU parallelization. In Chapter 4, we delve into the methodology applied to implement ICPCG on contemporary NVIDIA mobile GPUs using MATLAB PCT as well as supplementary implementations. Chapter 5 unveils the experimental results and findings obtained through the methodology, followed by a thorough discussion of these outcomes. Chapter 6 encapsulates the conclusion drawn from this study and outlines prospects for future research. Finally, the appendix furnishes additional information on the specifications of the GPUs and CPUs subjected to testing.

Chapter 2

Conjugate Gradients

The CG method is one of many common iterative methods used for solving large systems of linear equations that are symmetric, positive definite (SPD). It was developed by E.Stiefel and M.R. Hestenes [7]. This method is effective in solving a system, $Ax = b$, of n simultaneous equations in n unknowns, particularly if n is large. The matrix A is symmetric if $A = A^T$ and positive definite (PD) if $x^T Ax > 0$, for all $x \neq 0$ [4]. CG is considered to be a machine method as it has the following properties [7]:

1. Simplicity and minimal storage: The method is straightforward, consisting of repetitive elementary operations that demand minimal storage space.
2. Convergence and finite steps: The method is designed to converge rapidly, and ideally, it should reach a solution in a finite number of steps, even when infinite steps are theoretically required. A method that guarantees finite-step solutions, provided no rounding-off errors occur, is preferred.
3. Rounding-off error stability: The method maintains stability with respect to rounding-off errors. If necessary, it includes subroutines to ensure this stability. Rounding-off errors can be reduced by repeating the same routine, using the previous results as a refined estimate of the solution.
4. Progressive estimation: At each step, the method provides information about the solution, yielding a more accurate estimate than the previous one.
5. Utilization and original data: The method makes the most use of the initial data at each step of the routine. Special properties inherent to the given linear system, such as the presence of numerous zero coefficients, are preserved. (In contrast, certain methods like Gauss elimination may inadvertently alter these special properties.)

However, to grasp the method of CG, it is essential to have a prior understanding of both the steepest descent method and the method of conjugate directions.

2.1 The Method of Steepest Descent

In the method of steepest descent, we start at an arbitrary point x_0 and proceed towards a minimum value of the function f, defined in Equation 2.1 [8]. We advance through a sequence of steps $x_1, x_2, \ldots$ until we reach the proximity to the solution x. In each step, we choose the direction in which the function f decreases most rapidly, which is the negative gradient of f, denoted as $-f'(x_i)$. This direction is defined by the equation $-f'(x_i) = b - Ax_i$ [8].

$$f(x) = \frac{1}{2}x^T Ax - b^T x + c \tag{2.1}$$

In addition, the error, $e_i = x_i - x$, serves as a vector indicating the deviation from the solution x. Conversely, the residual, $r_i = b - Ax_i$, signifies the extent of deviation from the correct value of b. We can view the residual, $r_i = -Ae_i$, as the result of transforming the error e_i by the matrix A, placing it in the same space as b. More importantly, r_i corresponds to $-f'(x_i)$, representing the direction of the steepest descent, $r_i = -f'(x_i)$ [9].

A line search is a process that selects α to minimize the function f along a line. According to the fundamental of calculus principles, α is chosen to minimize f when the directional derivative, denoted as $\frac{d}{d\alpha}f(x_1)$, equals zero. Applying the chain rule, we have $\frac{d}{d\alpha}f(x_1) = f'(x_1)^T \frac{d}{d\alpha}x_1 = f'(x_1)^T r_0$. Thus, to find the optimal α, one should ensure that r_0 and $f'(x_1)$ are orthogonal by setting the expression to zero [9, 10].

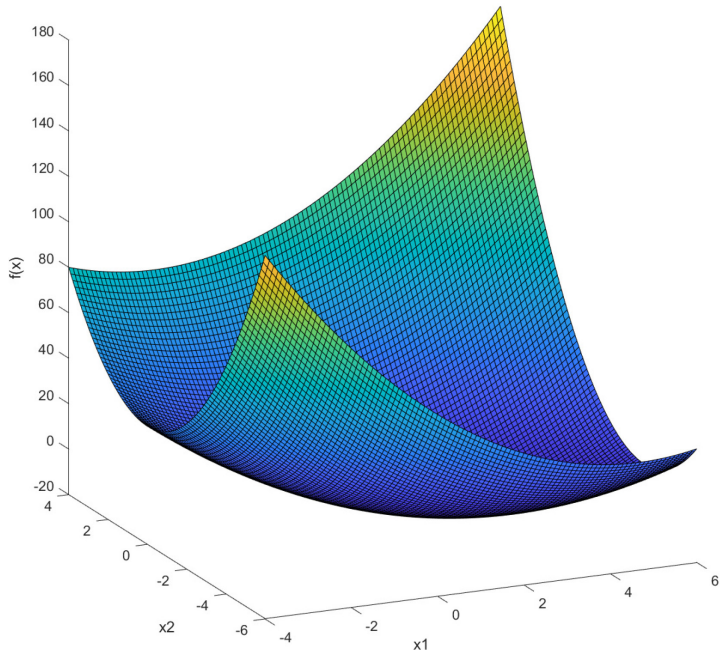

Figure 2.1: Surface of a quadratic form f(x).

In Figure 2.1, we visualize a representation of the surface of a quadratic function f. To demonstrate the application of a line search method, consider an initial point $x_0 = [-2, -2]^T$. Figure 2.2 depicts the intersection of a vertical plane with the paraboloid, while Figure 2.3 illustrates a search line along the contours of f. In Figure 2.2, the line search procedure aims to locate the point on the intersection of these two surfaces that minimizes the function f. At this specific point, the magnitude of the gradient vector, denoted as f', along the search line in Figure 2.3, reaches its maximum [10].

Consequently, the magnitude of the projection of the gradient vector onto the search line is zero. As we traverse along the search line, the magnitude of the gradient vector decreases, while the magnitude of the projection increases. This observation implies that at the minimum point on the search line, the gradient vector exhibits orthogonality with respect to the search line [9].

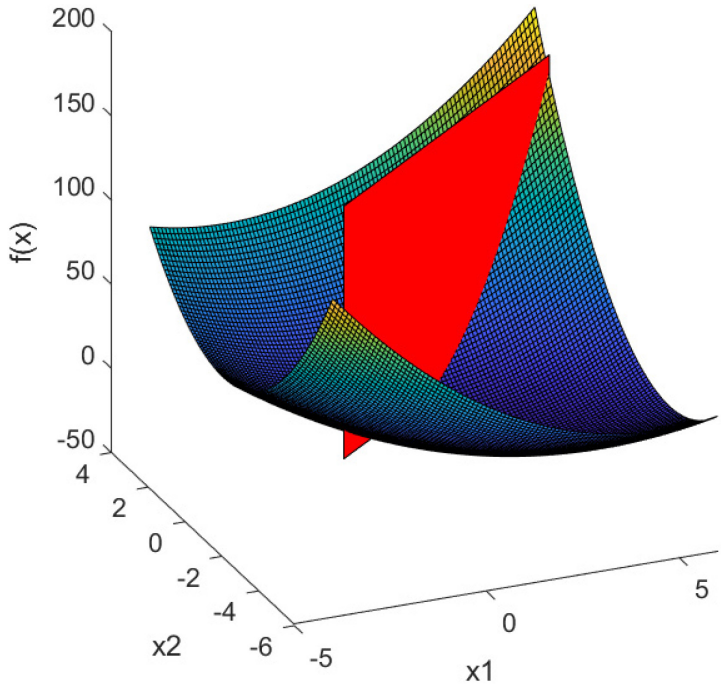

Figure 2.2: Line search on the quadratic form f(x).

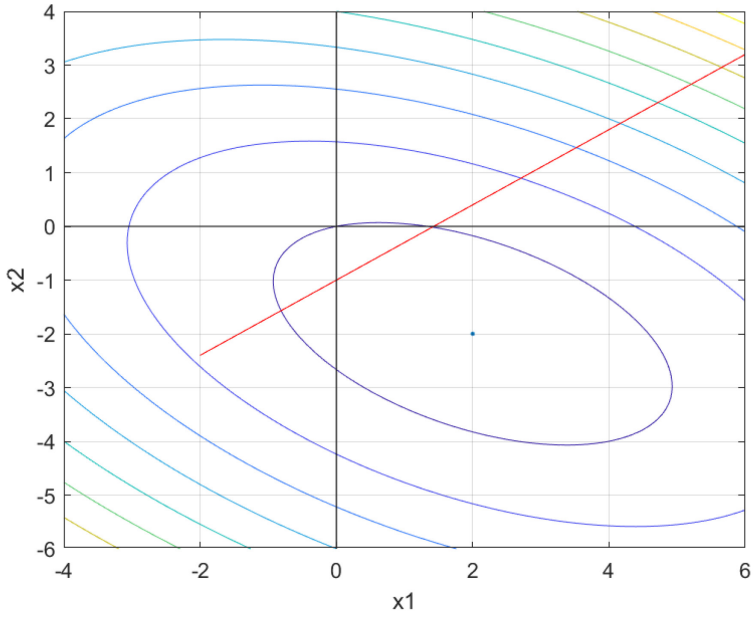

Figure 2.3: Contours of the quadratic form f(x) with the line search.

As per the previously mentioned definitions, the derivation of α leads to the formulation of the steepest descent method, as follows [9, 10]:

$$r_i = b - Ax_i \tag{2.2}$$

$$\alpha_i = \frac{r_i^T r_i}{r_i^T A r_i} \tag{2.3}$$

$$x_{i+1} = x_i + \alpha_i r_i \tag{2.4}$$

To optimize the computational efficiency by eliminating one of the two matrix-vector multiplications per iteration, we can pre-multiply Equation 2.4 by $-A$ and add b, resulting in a modified equation [9, 10]:

$$r_{i+1} = r_i - \alpha_i A r_i \tag{2.5}$$

While this modification reduces the number of matrix-vector multiplications per iteration, it is important to note that the computation of r_0, as per Equation 2.2, is still required initially. Once r_0 is determined, Equation 2.5 can be applied in subsequent iterations. Furthermore, the product Ar only needs to be calculated once for both Equations 2.3 and 2.5 [9].

It is worth highlighting that due to the use of r_0 in Equation 2.5, there is a potential accumulation of floating-point roundoff errors that might prevent x_i from converging to the true solution x. Therefore, Equation 2.2 can be recomputed periodically, rather than in every iteration, to ensure the correct residual is obtained [9].

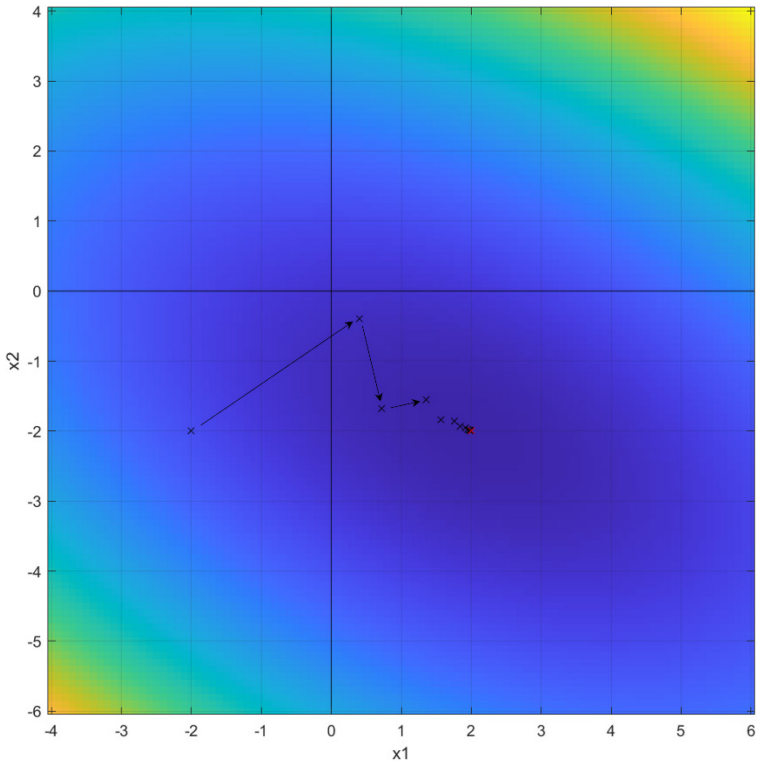

Figure 2.4: Meandering path of the method of steepest descent.

Upon applying the method of steepest descent using Algorithm 1 [10] on MATLAB, we can observe a meandering trajectory, which progressively converges toward the solution x, as depicted in Figure 2.4. Notably, each gradient vector is orthogonal to the preceding one. Furthermore, Algorithm 1 encompasses Equations 2.2 to 2.4 within its framework [10].

The convergence behaviour of the steepest descent method is characterized by the inequality $\|e_i\|_A \leq \left(\frac{\kappa-1}{\kappa+1}\right)^i \|e_0\|_A$, wherein κ represents the spectral condition number of matrix A in the linear system $Ax = b$. The spectral condition number κ indicates how sensitive the solution x is to small changes in the vector b, offering insights into the stability of the solution concerning perturbations in the right-hand side (RHS) vector [9, 11]. A higher condition number κ signifies a greater degree of ill-conditioning in the matrix. Unfortunately, this approach may lead to recurrent descent directions, resulting in an inefficient convergence process [8-10].

Algorithm 1 Steepest Descent Method

Given matrix A
Given RHS vector b
Initialize initial guess x_0
Set tolerance e and $i = 0$
At iteration i, while $\|r_i\| > e$, do
    $r_i = -\nabla f(x_i)$
    $\alpha_i = r_i^T r_i / r_i^T A r_i$ or perform an exact line search
    $x_{i+1} = x_i + \alpha_i r_i$
    $i = i + 1$
    if $\|r_i\| < e$ or $r_i = 0$
        return $[x_{i+1}, i]$
    end if
end while

2.2 The Method of Conjugate Directions

The method of conjugate directions refines the steepest descent method by incorporating a series of orthogonal search directions $d_0, d_1, \ldots, d_{n-1}$ to progress towards the minimum point. Within each of these search directions, the method takes a single step of precisely the correct length to align perfectly with the solution x. Once n such steps are executed, the solution x is determined [9].

These search directions exhibit A-orthogonality, meaning that two vectors, d_i and d_j, are considered A-orthogonal, or conjugate, if their dot product equals zero, $d_i^T A d_j = 0$. Furthermore, the current search direction d_i is A-orthogonal to the error of the subsequent iteration, labelled as e_{i+1}. This error is calculated as the difference between the point of the next iteration x_{i+1} and the true solution x. This orthogonality condition ensures that the method avoids retracing the same search direction as d_i, essentially equivalent to the process of seeking the minimum point along the search direction d_i, akin to the method of steepest descent [9].

Using the orthogonality of d_i and e_{i+1}, α is derived as [9]:

$$\alpha_i = -\frac{d_i^T A e_i}{d_i^T A d_i} \tag{2.6}$$

$$\alpha_i = \frac{d_i^T r_i}{d_i^T A d_i} \tag{2.7}$$

It is noteworthy that if we consider the search vector in Equation 2.6 as the residual, then Equation 2.7 would be identical to the formula employed in the steepest descent method [9].

To establish a set of search directions $d_0, d_1, \ldots, d_{n-1}$ that are A-orthogonal, the conjugate Gram-Schmidt process is used. By using a collection of n linearly independent vectors $u_0, u_1, \ldots, u_{n-1}$, we can derive d_i by subtracting the components in u_i that do not align with the A-orthogonal vectors from the previously determined d vectors [9].

In Figure 2.5(a), it shows that the conjugate Gram-Schmidt process initiated with two linearly independent vectors u_0 and u_1. Subsequently, in Figure 2.5(b), it designates d_0 to be u_0 and illustrates u_1 as a composed of two components: u^+ and u^*. Notably, the vector u^* is A-orthogonal, or conjugate, to d_0, while u^+ is parallel to d_0. Following this conjugation process, the A-orthogonal segment persists, resulting in the subsequent search direction d_1, as portrayed in Figure 2.5(c) [9].

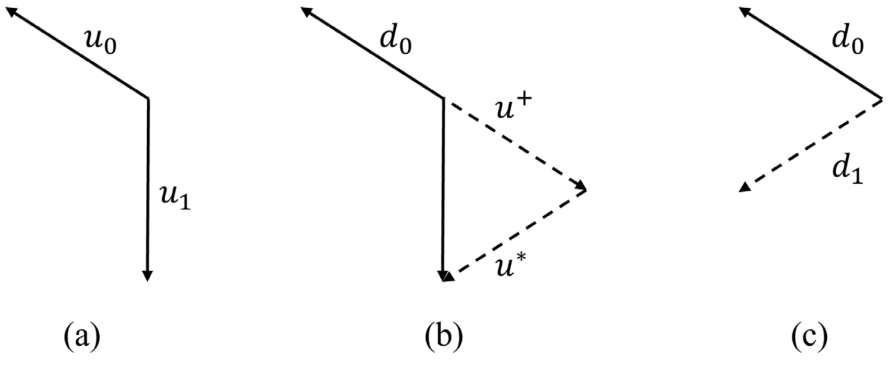

Figure 2.5: Gram-Schmidt conjugation of two vectors.

Generally, the process sets $d_0 = u_0$, and for the subsequent iterations $i > 0$, the search directions are [9]:

$$d_i = u_i + \sum_{k=0}^{i-1} \beta_{ik} d_k \tag{2.8}$$

In Equation 2.8, β_{ik} are defined for $i > k$, and using the definition of conjugacy, β_{ik} are derived as [9]:

$$\beta_{ik} = -\frac{u_i^T A d_j}{d_j^T A d_j} \tag{2.9}$$

Nonetheless, the Gram-Schmidt conjugation process within the method of conjugate directions necessitates the retention of all prior search vectors in memory for generating each new search vector. This incurs computational costs on the order of $O(n^3)$ to produce the complete set of search vectors [9]. Fortunately, when the search vectors are formulated by conjugating the axial unit vectors, the conjugate directions method aligns with the Gaussian elimination method. This equivalence is particularly evident in the method of conjugate gradients, where one concurrently executes the method of orthogonal directions within a scaled or stretched space [9].

Similar to Equation 2.5 in the method of steepest descent, the number of matrix-vector multiplications per iteration can be reduced by using a recurrence to find the residual, where $e_{i+1} = e_i + \alpha_i d_i$ [9]:

$$r_{i+1} = -Ae_{i+1} = r_i - \alpha_i A d_i \tag{2.10}$$

2.3 The Method of Conjugate Gradients

The method of conjugate gradients is essentially an adaptation of the method of conjugate directions, where the search directions are established by conjugating the residuals, achieved by setting $u_i = r_i$. Many of the properties found in the method of steepest descent and conjugate directions also apply to the CG method [9]. The motive of the CG method is the same as the steepest descent method, where the CG method minimizes the function f, as defined in Equation 2.1. In the CG method, the vectors are identified as: $\langle x_1, x_2, \dots, x_n \rangle = \langle d_0, d_1, \dots, d_{n-1} \rangle$, $\langle d_0, d_1, \dots, d_{n-1} \rangle = \langle r_0, r_1, \dots, r_{n-1} \rangle$, $\langle r_0, r_1, \dots, r_{n-1} \rangle = \langle b, Ab, \dots, A^{n-1}b \rangle$ [9]. Also, under the assumption of A being SPD, the A-norm is defined as $\|x\|_A = \sqrt{x^T A x}$. Moreover, since the search vectors are derived from the residuals, the subspace spanned by $\{r_0, r_1, \dots, r_{i-1}\}$ is identical to D_i. Each residual is orthogonal to the preceding search directions, which also happen to be the prior residuals: $r_i^T r_j = 0$, for $i \neq j$ [4, 9].

This concept is visually demonstrated in Figure 2.6, where a clear pattern emerges [9]. Each new residual r_i maintains orthogonality with respect to all prior residuals and search directions. Similarly, each new search direction d_i is purposefully constructed to be A-orthogonal to all the preceding residuals and search directions. Furthermore, the endpoints of r_2 and d_2 lie on a plane that runs parallel to the subspace D_2, and d_2 is a linear combination of r_2 and d_1 [9].

Referring to Equation 2.10, it becomes apparent that each successive residual r_i can be expressed as a linear combination of the previous residual and Ad_{i-1} [9]. By applying the definition that d_{i-1} belongs to D_i, written as $d_{i-1} \in D_i$, it follows that each subsequent subspace D_{i+1} is constructed by extending the previous subspace D_i with the subspace AD_i. As a result, the subspace D_i takes the following form [9]:

$$\begin{aligned} D_i &= \text{span}\{d_0, Ad_0, A^2 d_0, \dots, A^{i-1} d_0\} \\ D_i &= \text{span}\{r_0, Ar_0, A^2 r_0, \dots, A^{i-1} r_0\} \end{aligned} \tag{2.11}$$

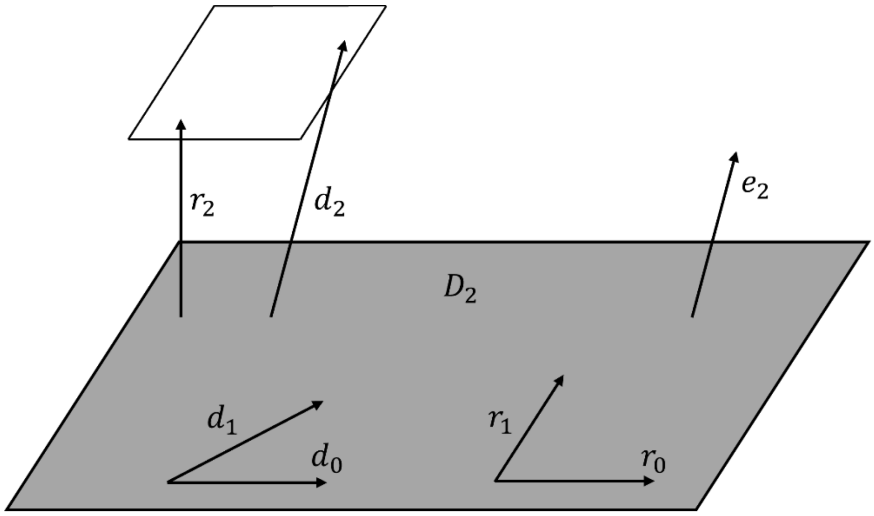

Figure 2.6: An illustration of the CG method.

The subspace, as defined in Equation 2.11, is commonly referred to as a Krylov subspace, which is a subspace of a vector space generated by iteratively applying a matrix to an initial vector that is the matrix A to the initial residual vector r_0 [9]. An essential property of a Krylov subspace is that the next residual vector r_{i+1} exhibits orthogonality with respect to D_{i+1}. In practical terms, this means that r_{i+1} is already A-orthogonal to D_i. This characteristic simplifies the Gram-Schmidt conjugation process because r_{i+1} is inherently A-orthogonal to all the preceding search directions [9].

The Gram-Schmidt conjugation process no longer necessitates the storage of previous search vectors to maintain the A-orthogonality of new search vectors. For this reason, this leads to a reduction in both space complexity and time complexity per iteration, from $O(n^2)$ to $O(m)$, where m represents the number of nonzero entries in the matrix A [9].

To summarize the CG method's workflow [9, 11]:

$$d_0 = r_0 = b - Ax_0 \tag{2.12}$$

$$\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i} \tag{2.13}$$

$$x_{i+1} = x_i + \alpha_i d_i \tag{2.14}$$

$$r_{i+1} = r_i - \alpha_i A d_i \tag{2.15}$$

$$\beta_{i+1} = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i} \tag{2.16}$$

$$d_{i+1} = r_{i+1} + \beta_{i+1} d_i \tag{2.17}$$

When we apply the CG method using Algorithm 2 [11] to the same example showcased in Figure 2.1 to 2.4 within the MATLAB environment, we observe a notably quicker convergence. This is characterized by the absence of a zigzagging trajectory toward the solution x, as depicted in Figure 2.7. Additionally, Algorithm 2 incorporates Equations 2.12 to 2.17 [9, 11].

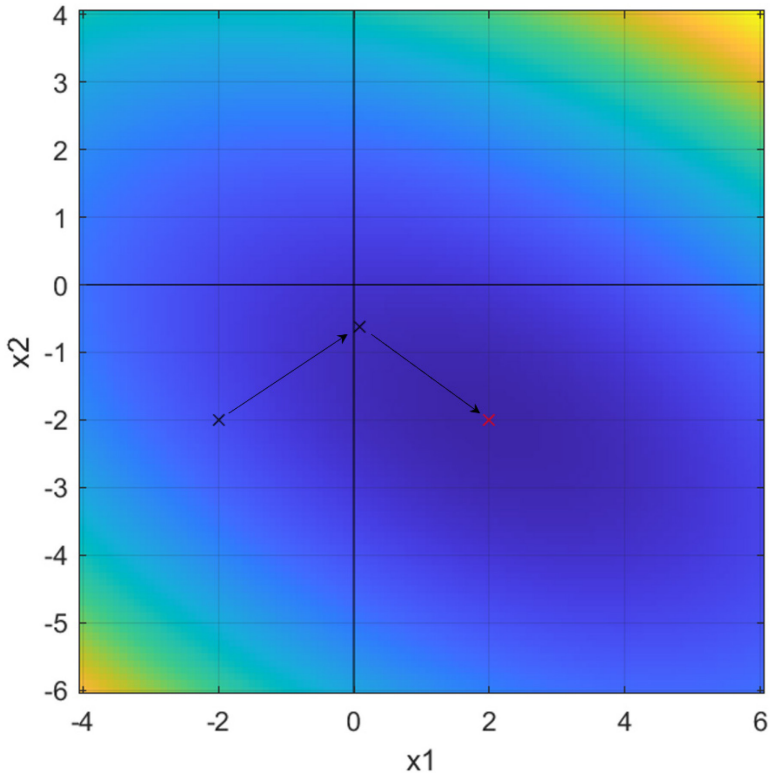

Figure 2.7: Direct path of the method of conjugate gradients.

Algorithm 2 Conjugate Gradients Method

Given matrix A
Given RHS vector b
Initialize initial guess x_0
Set tolerance e and maximum iterations N_{max}
Compute $r_0 = b - Ax_0$
Set $d_0 = r_0$
for iterations $i = 0: N_{max}$, do
    $\alpha_i = r_i^T r_i / d_i^T A d_i$
    $x_{i+1} = x_i + \alpha_i d_i$
    $r_{i+1} = r_i - \alpha_i A d_i$
    $\beta_{i+1} = r_{i+1}^T r_{i+1} / r_i^T r_i$
    $d_{i+1} = r_{i+1} + \beta_{i+1} d_i$
    if $\|r_{i+1}\| < e$, then
        return $[x_{i+1}, i]$
    end if
end for
Print failure to converge message when iteration $i > N_{max}$
return $[x_{N_{max}}, i = -1]$

As previously mentioned, the CG method theoretically converges after n iterations. However, in practical applications where n is typically large, performing n iterations become infeasible. In real-world scenarios, accumulating floating-point errors can lead to gradual loss of accuracy in the residual and a reduction in the A-orthogonality of the search vectors [12]. Thus, expecting an exact algorithm is not realistic.

When the CG method is applied to an SPD system $Ax = b$, the A-norms of the errors adhere to the inequality $\frac{\|e_i\|_A}{\|e_0\|_A} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i$, where κ represents the spectral condition number of matrix A. This inequality is derived from Chebyshev polynomials [12]. Similar to the method of steepest descent, if $\sqrt{\kappa}$ is relatively small, the CG method converges rapidly, while for a large $\sqrt{\kappa}$, convergence is slower. The value of κ depends on the spread between the largest and smallest

eigenvalues of A. When these eigenvalues are closely clustered, the CG method exhibits good convergence. Conversely, if the eigenvalues of A are widely separated, convergence is slower [12].

Each iteration of the CG method necessitates $O(n^2)$ floating-point operations, so executing n iterations result in a computational cost of $O(n^3)$ operations, which is equivalent to Cholesky decomposition. In practical implementation, convergence is often achieved in fewer than n iterations when using floating-point arithmetic. In general, the CG method outperforms the steepest descent method [12].

While the CG method stands out as a highly efficient iterative approach, dense and poorly conditioned matrices can be equally effectively solved through direct factorization and backward substitution [12]. In cases where the matrices are not SPD, the CG method can still be employed by transforming the original equation from $Ax = b$ to $A^TAx = A^Tb$ [4]. However, it is worth noting that preconditioning these systems can be challenging. Since this thesis primarily concentrates on SPD matrices, non-SPD matrices will not be explored in further detail.

2.4 Preconditioning

Preconditioning is a technique implemented to enhance the efficiency and robustness of iterative methods, such as the CG method. It accomplishes this by improving the condition number of a matrix [12]. Essentially, it transforms the original linear system $Ax = b$ into an equivalent system with the same solution that is easier to solve with an iterative solver. This transformation is achieved by left- or right- multiplying the system with a preconditioning matrix M. The preconditioner M needs to fulfill several criteria, including [12]:

1. It should be cost-effective to construct.
2. It should have a straightforward and efficient inversion process.
3. It should approximate A in a way that the product of M^{-1} and A is near to the identity matrix I and is non-singular.
4. The preconditioned system should be easier to solve with improved accuracy.

If the preconditioner M is applied to the left, the resulting system takes the form of Equation 2.19. Conversely, if M is applied to the right, it yields Equation 2.20. In the latter case, applying M to the right can be thought of as a change of variables $u = Mx$, and the system is then solved with respect to the unknown u [11]. In this thesis, we will focus on the left-multiplying preconditioner.

$$M^{-1}Ax = M^{-1}b \tag{2.19}$$

$$AM^{-1}u = b, x \equiv M^{-1}u \tag{2.20}$$

When $\kappa(M^{-1}A)$ is significantly smaller than $\kappa(A)$, or when the eigenvalues of $M^{-1}A$ exhibit better clustering than those of A, the iterative solution of Equation 2.19 can be achieved more rapidly than solving the original problem. The solution depends on the coefficient matrix $M^{-1}A$ instead of A [9, 12].

However, it is important to note that $M^{-1}A$ is not inherently symmetric or definite, even if both M and A possess these properties. This difficulty can be circumvented by recognizing that for every SPD M, there exists a matrix E that may not be unique with the property where E times its transpose equal to M, which is $EE^T = M$ [9]. This matrix E can be obtained through various methods, including Cholesky factorization. Importantly, the matrices $M^{-1}A$ and $E^{-1}AE^{-T}$ share the same eigenvalues λ, because if ν is an eigenvector of $M^{-1}A$ with the eigenvalue λ, then $E^T\nu$ is also an eigenvector of $E^{-1}AE^{-T}$ with the same eigenvalue λ [9].

The system $Ax = b$ can be transformed into the problem expressed in Equation 2.21. In this formulation, $\hat{x}$ is solved first followed by x. Notably, as $E^{-1}AE^{-T}$ is SPD, the method of steepest descent or CG can be used to solve for $\hat{x}$. The process of using the CG method to solve this system is also known as the transformed preconditioned conjugate gradient (TPCG) method [9].

$$E^{-1}AE^{-T}\hat{x} = E^{-1}b, \hat{x} = E^Tx \tag{2.21}$$

Evaluating the TPCG method reveals an undesirable characteristic—namely, the need to compute E. To address this, E can be eliminated via variable substitution, leading to the untransformed preconditioned conjugate gradient (UPCG) method [9]:

$$r_0 = b - Ax_0 \tag{2.22}$$

$$d_0 = M^{-1}r_0 \tag{2.23}$$

$$\alpha_i = \frac{r_i^T M^{-1} r_i}{d_i^T A d_i} \tag{2.24}$$

$$x_{i+1} = x_i + \alpha_i d_i \tag{2.25}$$

$$r_{i+1} = r_i - \alpha_i A d_i \tag{2.26}$$

$$\beta_{i+1} = \frac{r_{i+1}^T M^{-1} r_{i+1}}{r_i^T M^{-1} r_i} \tag{2.27}$$

$$d_{i+1} = M^{-1}r_{i+1} + \beta_{i+1} d_i \tag{2.28}$$

The effectiveness of a preconditioner M is primarily determined by the condition number of $M^{-1}A$, and, in some cases, the eigenvalue distribution within this transformed matrix [9]. As there are many ways to find M, the thesis will mainly focus on IC factorization technique.

2.4.1 Incomplete Cholesky Factorization

The incomplete Cholesky factorization is a fundamental technique in numerical linear algebra. Generally, the IC factorization is similar to Cholesky factorization, except the former is designed for sparse matrices. It is a variant that approximates the Cholesky factorization of a sparse matrix without filling in zero-fill-ins or minimal fill-ins whenever possible, which makes the IC factorization more memory-efficient as the factorization matrix remains its sparsity [3].

Cholesky factorization is applied to decompose a real SPD matrix A into the structure shown in Equation 2.29, where L represents a lower triangular matrix. The computation of the elements within L can be performed column by column, following recursive equations like Equation 2.30 for diagonal elements and Equation 2.31 for elements below the diagonal [3]. Given that L is lower triangular, it simplifies the computation of its inverse L^{-1} and the inverse of its transpose $(L^T)^{-1}$. This enables the solution of the linear system $Ax = b$, where the process first computes y through forward elimination and then determines x using backward substitution, as illustrated in Equation 2.32 [3].

$$A = LL^T \tag{2.29}$$

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2} \tag{2.30}$$

$$L_{ji} = \frac{A_{ji} - \sum_{k=1}^{i-1} L_{jk}L_{ik}}{L_{ii}} \tag{2.31}$$

$$j = (i+1), (i+2), \dots, n$$

$$y = L^{-1}b,\ x = (L^T)^{-1}y \tag{2.32}$$

In terms of computational complexity, the Cholesky factorization has a cost of $O(n^3)$ and involves computing n square roots. Overall, Cholesky factorization tends to be approximately twice as fast as the lower-upper (LU) decomposition when applied to a PD matrix [12].

2.4.2 Incomplete Cholesky Preconditioned Conjugate Gradient

To implement the ICPCG method, assume the existence of a PD preconditioner M, which can be decomposed into an IC factorization $M = LL^T$, where L is a lower triangular matrix. This factorization serves the purpose of preserving symmetry using the split preconditioning approach and results in an equivalent system $\bar{A}\bar{x} = \bar{b}$. This system yields the SPD matrix $\bar{A}$, as shown in Equation 2.33 [11, 12], resembling Equation 2.21. Initially, the CG method is applied to solve for $\bar{x}$ in $\bar{A}\bar{x} = \bar{b}$, followed by solving for x in $\bar{x} = L^T x$ [3] .

$$\begin{gathered}(L^{-1}AL^{-T})L^T x = L^{-1}b \\ \bar{A} = L^{-1}AL^{-T},\ \bar{x} = L^T x,\ \bar{b} = L^{-1}b\end{gathered} \tag{2.33}$$

However, the Cholesky factor L in Equation 2.33 is often less sparse than M. Therefore, L might be constrained to maintain the same pattern of nonzero elements [3]. When an element a_{ij} off the diagonal of A is zero, the corresponding element l_{ij} is also set to zero. Consequently, L retains the same distribution of nonzero values as A below the diagonal elements; hence, it is an incomplete factorization. With this adjustment, M takes the form shown in Equation 2.34, where E represents a small error matrix containing nonzero entries exclusively in the elements that have been forced to zero [3, 12].

$$M = LL^T + E \tag{2.34}$$

During the algorithm's execution, it is critical that all L_{ii} values are greater than zero. If L_{ii} equals zero, the algorithm will fail. Similarly, if L_{ii} is less than zero, then LL^T is not PD, which implies that the CG method cannot provide an exact solution. A complete Cholesky factorization will always yield L_{ii} values greater than zero. Additionally, it has been proven that if A is an A-matrix, i.e., $A_{ij} \leq 0$ if $i \neq j$, the IC factorization will consistently yield L_{ii} values greater than zero [3].

Algorithm 3 Incomplete Cholesky Factorization

Given matrix A
function $L = \text{icholesky}(A)$
  for iterations $i = 1{:}\, n$, do
    $\text{temp} = A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2$
    if $\text{temp} \leq 0$, then
      Print error messages
      return
    end if
    $L_{ii} = \sqrt{\text{temp}}$
    for iterations $j = i + 1{:}\, n$, do
      if $A_{ji} == 0$, then
        $L_{ji} = 0$
      else
        $L_{ji} = \left(A_{ji} - \sum_{k=1}^{i-1} L_{jk} L_{ik}\right)/L_{ii}$
    end for
  end for
end function

Algorithm 3 represents the IC factorization, which integrates Equations 2.29 to 2.31 [3, 12]. An adjustment is made in Lines 11 to 15, ensuring that when a_{ij} equals zero, the corresponding element l_{ij} is forced to zero, ultimately resulting in the Cholesky factor L. Subsequently, Algorithms 4 and 5 delineate the forward elimination and backward substitution, respectively. Algorithm 6 combines the functionalities of Algorithms 4 and 5 into a unified function. Finally, Algorithm 7 encapsulates the ICPCG method, which incorporates Equations 2.22 to 2.28. However, a successful convergence will depend on how good an approximate inverse $(LL^T)^{-1}$ is [3, 12].

Algorithm 4 Forward Elimination

1: Given lower-triangular matrix L
2: Given RHS vector b
3: function $y = \text{forward}(L, b)$
4: for iterations $i = 1: n$, do
5: for iterations $j = 1: i - 1$, do
6: $\text{temp} = \sum_{j=1}^{i-1} L_{ij}\, y_j$
7: end for
8: $y_i = (b_i - \text{temp})/L_{ii}$
9: end for
10: end function

Algorithm 5 Backward Substitution

Given upper-triangular matrix U
Given vector y
function $x = \text{backward}(y, U)$
    for iterations $i = n:1$, do
        for iterations $j = i + 1:n$, do
            $\text{temp} = \sum_{j=i+1}^{n} U_{ij} x_j$
        end for
        $x_i = (y_i - \text{temp})/U_{ii}$
    end for
end function

Algorithm 6 Solve Cholesky

Given lower-triangular matrix L
Given RHS vector b
function $x = \text{cholsolve}(L, b)$
    $y = \text{forward}(L, b)$
    $x = \text{backward}(y, L^T)$
end function

Algorithm 7 Incomplete Cholesky Preconditioned Conjugate Gradients Method

Given matrix A
Given RHS vector b
Initialize initial guess x_0
Set tolerance e and maximum iterations N_{max}
function $pcg(A, b, x_0, e, N_{max})$
    $L = icholesky(A)$
    $r_0 = b - Ax_0$
    $z_0 = cholsolve(L, r_0)$, let $z_i = (LL^T)^{-1} r_i$
    $d_0 = z_0$
    for iterations $i = 0: N_{max}$, do
        $\alpha_i = z_i^T r_i / d_i^T$
        $x_{i+1} = x_i + \alpha_i d_i$
        $r_{i+1} = r_i - \alpha_i A d_i$
        if $\|r_{i+1}\| < e$, then
            return $[x_{i+1}, i]$
        end if
        $z_{i+1} = cholsolve(L, r_{i+1})$
        $\beta_{i+1} = r_{i+1}^T z_{i+1} / r_i^T z_i$
        $d_{i+1} = z_{i+1} + \beta_{i+1} d_i$
    end for
    Print failure to converge message when iteration $i > N_{max}$
    return $[x_{N_{max}+1}, i = -1]$
end function

It has been observed that the IC preconditioning may encounter stability issues, especially in challenging scenarios where cancellation errors occur. To enhance the algorithm's reliability, the drop tolerance-based IC factorization method is adopted [12]. This method retains the off-diagonal elements computed by the Cholesky algorithm if a specific condition is met, and otherwise, it preserves the original values, as shown in Equation 2.35. As the drop tolerance decreases, the IC factor tends to become denser [12].

$$L_{ji} = \begin{cases} \dfrac{A_{ji} - \sum_{k=1}^{i-1} L_{ji} L_{ik}}{L_{ii}} & A_{ji}^2 > e^2 A_{jj} b_{ii} \\ A_{ji} & \text{otherwise} \end{cases} \tag{2.35}$$

Chapter 3
Graphics Processing Units

General-purpose Graphics Processing Units (GPGPUs) are specialized hardware originally designed for rendering graphics, including computations for both geometry (vertices) and rasterization (pixels), but have evolved to excel in parallel processing tasks. The idea of using GPUs for non-graphical computation began to gain traction in the early 2000s [2]. These GPUs are equipped with thousands of small processing cores optimized for parallelism, making them suitable for a wide range of computational workloads [2].

3.1 Graphics Processing Unit vs Central Processing Unit

CPUs are characterized as latency-oriented processors designed for task parallelism. They allocate a substantial number of transistors for caching and employ sophisticated flow control mechanisms. Modern CPUs can be considered multicore processors as they can achieve their maximum performance potential with just a few threads [13]. In contrast, GPUs are highly throughput-oriented processors with a focus on data parallelism. They efficiently manage the relatively expensive global memory accesses by leveraging a multitude of parallel threads. GPUs are manycore processors, and they require a large number of threads, often in the thousands, to operate at their full capacity. This makes GPUs have larger memory bandwidth but higher memory latency, whereas CPUs have lower latency but lower bandwidth [13].

However, the significant boost in throughput facilitated by a GPU does come with some trade-offs. One of the primary concerns is the potential bottleneck in memory access during calculations. Before performing calculations, data must be transferred from the host, CPU, to the device, GPU, and afterward, it needs to be retrieved. Since a GPU is connected to the CPU through the Peripheral Component Interconnect Express (PCIe) bus, memory access tends to be slower compared to traditional CPUs. Hence, the overall acceleration in computational speed is constrained by the amount of data transfer that takes place within the algorithm [1].

GPUs are typically employed as coprocessing units alongside CPUs and are particularly well-suited for tasks that involve high regularity and significant arithmetic intensity. Typically, a CPU handles the sequential parts of a program, while a GPU takes care of the computationally intensive portions to accelerate overall processing speed. Moreover, GPUs require explicit parallel programming using an API, such as NVIDIA CUDA or OpenCL, while CPUs are programmed using traditional languages, such as C++ [13].

3.2 Graphics Processing Unit Architecture

Contemporary GPUs are composed of various components, and specific GPU models may use varying nomenclature for these constituents. The key constituents are Streaming Multiprocessors (SMs), memory hierarchy, Single Instruction Multiple Data (SIMD) stream paradigm, rendering pipelines, memory controllers, display output capabilities, interconnects, and unified memory [6]. The presence and configuration of these components can differ across GPU models. In the context of the GPUs examined in this thesis, we will mainly focus on GPUs from NVIDIA.

SMs serve as the CPU of the GPU, responsible for executing the core computation. NVIDIA GPUs are equipped with NVIDIA CUDA Cores, specifically designed to accelerate general-purpose computing tasks, including matrix operations [14]. An SM is engineered to execute hundreds of threads simultaneously. These threads function as parallel processors, handling floating-point mathematical operations. To efficiently manage this multitude of threads, it employs a unique architecture known as Single-Instruction, Multiple-Thread (SIMT). This architecture pipelines instructions, exploiting both instruction-level parallelism within a single thread and extensive thread-level parallelism through simultaneous hardware multithreading [6]. All data processed by a NVIDIA GPU is channeled through threads, and each thread possesses its own memory register that is inaccessible to other threads [6].

Furthermore, the concept of a CUDA block, also known as a thread block, entails the grouping of threads, which are further organized into a grid. A kernel is executed as a grid of blocks of threads. Thread blocks are required to execute independently in any sequence, either serially or concurrently. Each thread block is managed by one SM, and an SM can handle multiple concurrent thread blocks based on the resources needed by those blocks [6]. This logical arrangement

enhances the efficiency of data mapping. Importantly, thread blocks share memory on a per-block basis, implying that every thread within a specific CUDA block can access the same shared memory. In the current CUDA architecture, each block consists of 1024 threads [6].

Kernel grids play a role in grouping thread blocks under the same kernel. The thread blocks can be arranged in one-dimensional, two-dimensional, or three-dimensional grids, as shown in Figure 3.1, facilitating parallel execution, especially for tasks demanding more than 1024 threads. However, the synchronization that occurs at the block-level does not extend to the grid-level as the shared memory is inaccessible to different thread blocks [6]. Lastly, there is an optional hierarchy level known as Thread Block Clusters (TBC), comprising thread blocks. Thread blocks within a TBC are guaranteed to be scheduled together on a GPU processing cluster, akin to how threads within a thread block are ensured to be co-schedule on an SM. This hierarchy is illustrated in Figure 3.2 [6].

GPUs feature diverse memory levels, and memory allocation adheres to a specific hierarchy within CUDA, as depicted in Figure 3.3 [6]. This hierarchy is managed automatically by CUDA compiler or can be manually configured by developers to optimize memory utilization. The memory levels are registers, read-only (RO) memory, L1 cache/shared memory, L2 cache, and global memory [6].

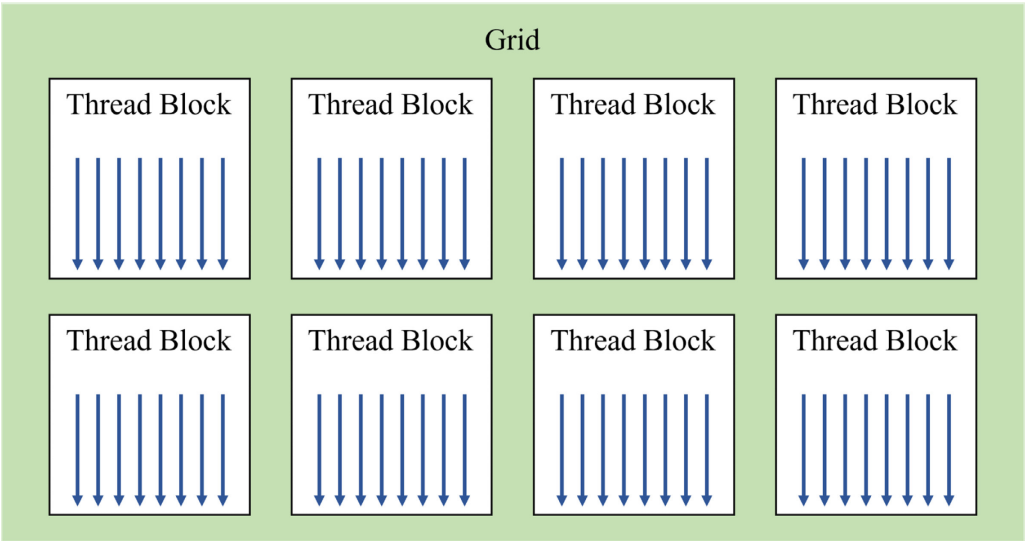

Figure 3.1: Grid of Thread Blocks.

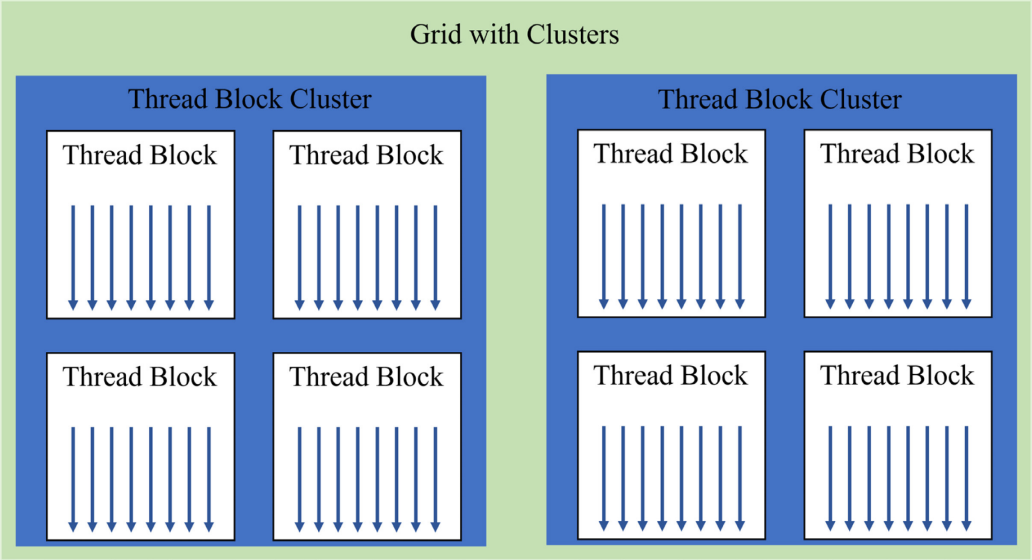

Figure 3.2: Grid of Thread Block Clusters.

In broad terms, registers are assigned to threads, and data stored in registers can be processed at an accelerated rate compared to other data storage locations. RO memory spaces, i.e., the constant and texture memory spaces, are accessibly by all threads. They are situated on-chip within SMs and serve specific functions like texture memory [6]. The global, constant, and texture memory spaces are optimized for different memory usages and are persistent across kernel launches by the same application. It is more efficient to access data from RO memory than to resort to global memory [6].

L1 cache/shared memory is on-chip memory that is shared among thread blocks, with its management being a combined effort between hardware and software. Thread blocks in a TBC can perform read, write, and atomics operations on each other's shared memory [6]. Similarly, as it is on-chip, the L1 cache/shared memory offers faster access speeds compared to L2 cache and global memory. L2 cache stores both global and local memory and is accessible to all threads across all thread blocks. Retrieving data from L2 cache is faster than fetching it from global memory [6]. Finally, global memory corresponds to a dynamic random-access memory (DRAM) and is comparable to random-access memory (RAM) in CPU. All threads have access to the same global memory, but global memory inherently operates at a slower speed than L2 cache [6].

Modern GPUs predominantly embrace a SIMD stream architecture, characterized by a single control processor and instruction memory. Within this architecture, a solitary instruction is

replicated and executed simultaneously across all threads at any given moment, enabling efficient data parallelism [6]. In the case of NVIDIA GPUs, they also employ the SIMT model to effectively manage their extensive thread pool. SIMT is an enhancement of the SIMD model by introducing multithreading. This addition enhances overall efficiency by reducing the overhead related to instruction fetching. Consequently, SIMT empowers developers to craft code that exhibits thread-level parallelism for independent, scalar, threads, as well as data-parallel code for coordinated threads [6].

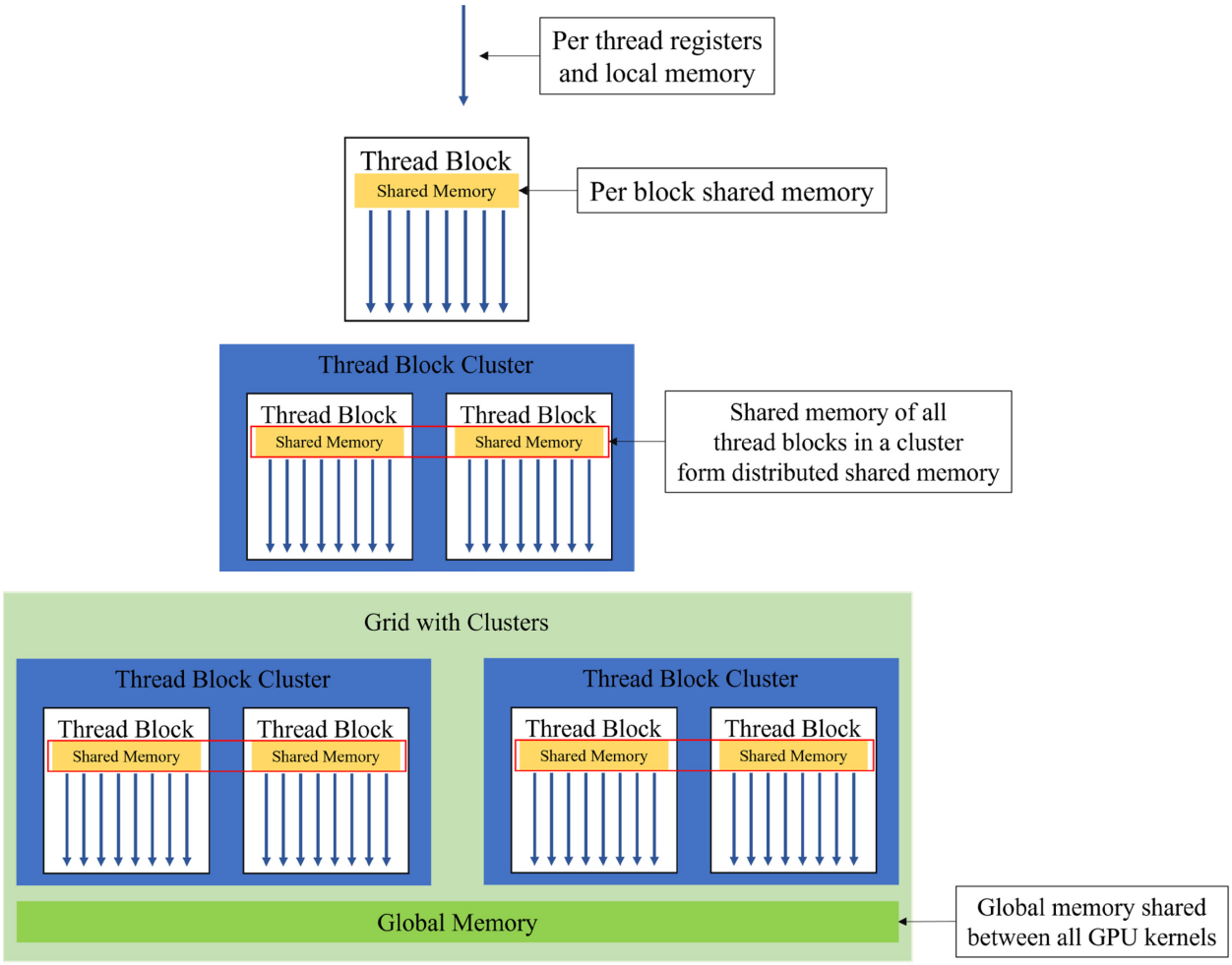

Figure 3.3: Memory Hierarchy.

3.3 MATLAB Parallel Computing Toolbox

The Parallel Computing Toolbox (PCT), developed by MATLAB, provides a platform for tackling computationally and data-intensive tasks by harnessing the power of multicore processors, GPUs, and computer clusters [5, 15]. Unlike some other tools, it does not necessitate the use of APIs like CUDA to fully utilize the computing potential, whether it is multiple GPUs on a desktop, computer clusters, or cloud environments. It seamlessly integrates with parallel-enabled functions in MATLAB and various other toolboxes [5, 15].

In contrast to CUDA, MATLAB PCT simplifies parallelization by abstracting the low-level coding required. While CUDA often demands developers to write code at a lower level to achieve parallelism, MATLAB PCT automates the parallelization of the PCG algorithm on the GPU using parallel-enabled functions found in the toolbox. This enables developers to utilize the GPU's parallel processing capabilities without the need for explicit parallelization implementation [5, 15]. However, the effectiveness of GPU acceleration using MATLAB PCT may vary depending on factors such as the specific problem being solved and the hardware configuration. Its high-level programming may not always optimize performance as effectively as manually optimized CUDA code [5, 15].

On the other hand, similar to CUDA, PCT taps into NVIDIA GPUs for both non-graphics and graphics computations, all within the MATLAB programming language. This eliminates the need to switch to a different programming language, allowing developers to concentrate on their applications rather than getting bogged down in performance optimization. Moreover, for those who prefer CUDA, MATLAB can interface with CUDA code, enabling the execution of CUDA operations alongside PCT [5]. For these reasons, MATLAB PCT is used in this study to evaluate the performance of its functions.

Chapter 4
Methodology

In this chapter, the methodologies and techniques applied throughout the research are comprehensively presented. The objective is to provide a clear understanding of the implemented methods. These methods encompass the implementation of the ICPCG method on different computing hardware and the exploration of additional methods. The inclusion of other methods serves to enhance the comprehension of the mobile GPU's performance. These supplementary methods involve exploring the performance of the backslash operator on both CPUs and mobile GPUs, assessing the data handling capabilities of mobile GPUs, and evaluating resource contention. These methods are tested on generated matrices, which are large and sparse, and specific types of problem with a distinguished pattern in the matrices, as further elaborated in the next chapter (Chapter 5). The methodologies discussed are important for assessing the performance, efficiency, and scalability of these techniques, leading to valuable insights into their real-world applications.

4.1 ICPCG on CPUs and GPUs with Specific Problem Types

The examination of the ICPCG method encompasses two distinct sub-sections: the ICPCG methodology on CPUs and ICPCG methodology on GPUs. These sub-sections offer detailed insights into the benchmarking setups for ICPCG on each platform, leveraging the capabilities provided by MATLAB PCT [5]. Specifically, the ICPCG method is assessed using specific problem types characterized by unique matrix patterns. This deliberate choice facilitates a comprehensive comparison of ICPCG's performance and scalability across diverse computing environments, shedding light on its behaviour on both CPU and GPU architecture.

4.1.1 ICPCG on CPUs and GPUs using Parallel Computing Toolbox Commands

The evaluation commences by loading a specific problem type into the workspace of MATLAB. To scrutinize the ICPCG method's performance, the GPU undergoes initialization, and its memory is cleared using the `gpuDevice` [16] and `reset` functions, respectively, from MATLAB PCT. Subsequently, the IC decomposition is applied to the matrix `A` of the loaded problem, utilizing the `ichol` function [17] on the CPU. This function takes matrix `A` as input and yields `L1` as output, and the time taken for the IC decomposition is recorded.

Once the IC decomposition concludes, the PCG method is employed using the `PCG` function [18]. This function incorporates multiple inputs, including matrix `A` with `N`-by-`N` dimension, RHS column vector `b` with `N`-by-`1` dimension, tolerance, maximum number of iterations, preconditioners `M1` and `M2`, and an initial guess `x0`. In this context, the predefined values for tolerance, maximum number of iterations, preconditioner, and initial guess are 10^{-5}, 10^{5}, the output matrix of the IC decomposition `L1`, and the zero vector, respectively. Matrix `A` and vector `b` are directly sourced from the loaded problem. The PCG method is then executed on the CPU, followed by a similar execution on the GPU. On the CPU, the PCG method is timed using `tic` and `toc` functions [19, 20].

For the GPU execution, matrix `A`, vector `b`, and preconditioner `L1` must be transferred from the host to the GPU using MATLAB PCT's `gpuArray` function [21]. Following the memory allocation, the PCG method is applied on the GPU using the `PCG` function [18], mirroring the CPU execution. With the exception that on the GPU, the PCG method is timed using both the `gputimeit` function [22], which is from MATLAB PCT, and the `tic` and `toc` functions [19, 20]. To utilize `gputimeit`, a function handle, `pcgFcn`, must be created first. A function handle is a data type in MATLAB that stores an association with a function, enabling the passing of a function to another function [23]. When employing `tic` and `toc` to measure execution time on the GPU, the `wait` function [24] must be applied to ensure accurate timing. This is necessary because the program must wait for operations to complete before calling `tic` and `toc`. The `gputimeit` function, unlike the `tic` and `toc` functions, ensures that all GPU operations have completed before timing and adjusts for any associated overhead. Therefore, when using the `tic` and `toc` functions,

the `wait` function must also be employed to ensure that all GPU operations have finished before recording the time. However, `tic` and `toc` do not consider the overhead. The PCG method is implemented several times to capture the best timing for both CPU and GPU executions [25].

It is crucial to note that the `PCG` function is fully supported by MATLAB PCT [18], ensuring smooth execution of the function with the aid of the toolbox without encountering potential errors, such as running out of memory. However, this support is not extended to the backslash operator, a point that will be further explained in the subsequent sub-chapter (Chapter 4.2.2).

Utilizing the execution timings of the PCG method on each platform, the floating-point operations per second (flops) are calculated for the corresponding hardware using the formula derived from the Linpack TPP benchmark of the HPC Challenge [26], as shown in Equations 4.1 and 4.2 where n represents the size of matrix A, i.e., an n-by-n matrix. The formulas measure the floating-point rate of execution, commonly known as flops, and incorporate a multiplier to yield gigaflops, specifically for solving a linear system of equations. The results are then returned as outputs of the ICPCG function.

$$\text{flops} = \frac{2}{3}n^3 + \frac{3}{2}n^2 \tag{4.1}$$

$$\text{gigaflops} = \text{flops} \div \text{execution time} \div 10^9 \tag{4.2}$$

Algorithm 8 delineates the assessment process for this section. In Lines 2 and 3, the GPU is reset. Lines 4 to 6 depict the IC decomposition process, while Lines 8 and 10 showcase the implementation of PCG on the CPU. Subsequently, Lines 11 to 13 illustrates the data transfer from the host to the GPU, and Lines 14 to 18 demonstrate the PCG being employed on the GPU using `tic` and `toc`. Furthermore, Lines 19 and 20 present the GPU's PCG execution using `gputimeit`, and Lines 21 and 22 outline the computation of gigaflops.

Algorithm 8 ICPCG function on CPUs and GPUs using MATLAB PCT

1: function iccg_pct(A, b)
2: $\text{gpu} = \text{gpuDevice}$
3: reset(gpu)
4: $t_{ichol} = \text{tic}$
5: $L1 = \text{ichol(A)}$
6: $t_{ichol} = \text{toc}$
7: Set tolerance e and maximum number of iterations N_{max}
8: $t_{PCG_{CPU}} = \text{tic}$
9: Execute PCG on CPU $\text{pcg}(A, b, e, N_{max}, L1)$
10: $t_{PCG_{CPU}} = \text{toc}$
11: gpuArray(A)
12: gpuArray(b)
13: gpuArray(L1)
14: Wait for operations before start timing wait(gpu)
15: $t_{PCG_{GPUtictoc}} = \text{tic}$
16: Execute PCG on GPU $\text{pcg}(A, b, e, N_{max}, L1)$
17: Wait for operations before stop timing wait(gpu)
18: $t_{PCG_{GPUtictoc}} = \text{toc}$
19: $\text{pcgFcn} = @(\,)\ \text{pcg}(A, b, e, N_{max}, L1)$
20: $t_{PCG_{GPUtimeit}} = \text{gputimeit(pcgFcn)}$
21: Calculate flops for CPU and GPU $\text{flops} = 2/3 * n^3 + 3/2 * n^2$
22: Convert to gigaflops for CPU and GPU $\text{gflops} = \text{flops}/t/10^9$
23: return $[t_{ichol}, t_{PCG_{CPU}}, t_{PCG_{GPUtictoc}}, t_{PCG_{GPUtimeit}}, \text{gflops}_{CPU}, \text{gflops}_{GPU}]$
24: end function

4.1.2 ICPCG on CPUs using Single Program Multiple Data Statements

Similar to the preceding section, the same specific problem type is initially loaded into the MATLAB's workspace. To implement the ICPCG method on CPUs utilizing single program multiple data (SPMD) statements [27], a feature introduced by MATLAB PCT, a parallel pool of workers or processes within a process-based environment must be established through the `parpool` function [28, 29]. If a parallel pool of workers already exists, it needs to be closed before creating a new one. When this function is utilized, MATLAB establishes a pool on the local machine, assigning one worker to each physical CPU core. These parallel workers are subsequently entrusted with computational tasks using the SPMD statements, enabling the execution of parallelized code on workers within the same multi-core CPU. The SPMD statement allows operations that are within the SPMD body to be performed on the parallel workers simultaneously. Each worker can operate on a different data set or different portion of the distributed data and can communicate with other parallel workers while performing the parallel computations [27].

After creating the parallel pool of workers, a function handle [23] is generated to pass the ICPCG function, `iccg`, to a timing function, `timingfcn`, where the execution time is recorded. The `iccg` function initiates the IC decomposition process followed by the PCG method, utilizing the `ichol` and `pcg` functions [17, 18], respectively, as presented in Algorithm 10. The `ichol` function takes matrix A from the loaded problem, producing L1 as the output matrix. The `pcg` function takes inputs such as matrix A and vector b from the loaded problem, the preconditioner L1, and preset values of 10^{-5} and 10^{5} for tolerance and the maximum number of iterations, respectively.

Within the timing function `timingfcn`, as shown in Algorithm 9, two inputs are taken: an input function (in this case, `iccg`) which is passed to another function via the function handle [23], and the number of parallel workers. While an output array of the best execution times for a given level of concurrency is returned. Using `timingfcn`, the input function, `iccg`, is invoked multiple times for each number of parallel workers and with each execution timed, all within the SPMD

statements [27]. After each parallel execution of the ICPCG method on the CPU, the timings are compared to record the best execution time for the specified number of parallel workers.

Upon completing the execution of the ICPCG method, the gigaflops are calculated using the formulas from the Linpack TPP benchmark of the HPC Challenge, outlined in Equations 4.1 and 4.2 [26]. Ultimately, the results are returned as outputs of the ICPCG on parallel workers function.

Algorithm 9 Timing Function

1: Specify function, f, as iccg and an array of number of parallel workers, $N_{workers}$
2: function $timingFcn(f, N_{workers})$
3: Initialize an array for time, t, and number of executions, N_{exe}
4: for iterations $i = 1: length(N_{workers})$, do
5: $n = N_{workers}(i)$
6: $spmd(n)$
7: Initialize $t_n = \infty$
8: for iterations $k = 1: N_{exe}$, do
9: labBarrier
10: $t_{current} = tic$
11: f( )
12: $t_{current} = gop(@max, toc)$
13: $t_n = \min(t_n, t_{current})$
14: end for
15: end spmd
16 $t(i) = t_n$
17: clear t_n k $t_{current}$
18: end for
19: return [t]
20: end function

Algorithm 10 ICCG Function

1: Specify tolerance e, maximum number of iterations, N_{max}
2: function iccg(A, b)
3: $L1 = ichol(A)$
4: $pcg(A, b, e, N_{max}, L1)$
5: end function

4.2 Backslash on CPUs and GPUs

In MATLAB, the backslash operator solves a system of linear equations, expressed as x = A\b [30]. In this phase of performance analysis, we conduct tests using the backslash operator, assessing its functionality on both CPUs and mobile GPUs. The insights derived from this evaluation will contribute to our understanding of how the mobile GPUs handle computationally demanding operations, including the ICPCG method. The evaluation encompasses its application to both randomly generated matrices and specific problem types characterized by distinctive patterns in the matrices. The detailed procedures for conducting this analysis are outlined, highlighting the matrix and vector sizes considered for the testing process.

4.2.1 A\b on CPUs and GPUs with Generated Matrices

To evaluate the backslash operator's performance [30] on CPUs and GPUs, it is imperative to first clear the GPU memory to ensure the optimal utilization for this analysis. Additionally, the GPU is initialized using the `gpuDevice` function [16], which belongs to the MATLAB PCT. After determining the available CPU and GPU memory in gigabytes (GB), an array of suitable sizes for matrix A is computed, considering both single- and double-precision elements with a specified fixed step size. This precaution prevents potential errors by ensuring that the generated matrices and vectors of varying dimensions do not exceed the available memory [31].

Subsequently, a versatile function is designed to generate matrix A and vector b based on the arrays of appropriate sizes for both single- and double-precision, applicable to both CPU and GPU. This function is separate from the test function to ensure that the recorded time excludes the cost associated with data transfer between CPU and GPU, the duration taken for matrix creation, or other parameters [31]. Matrix A is constructed with significantly larger diagonal elements than non-diagonal elements, emulating real-world scenarios. A dedicated test function is also established to execute x = A\b [30]. In this test function, the backslash operator is invoked multiple times to capture the optimal execution time for the given size and precision type. The test function remains mostly consistent for both CPU and GPU, although there is a variation in the GPU procedure [31]. For the GPU, the data must be transferred from the CPU to the GPU, facilitated by the function `gpuArray` from MATLAB PCT [21]. Furthermore, the test function accommodates the time required to introduce overhead, and this duration is subsequently subtracted from the execution time. This adjustment ensures that only the actual execution time is taken into account. Additionally, a wait function is crafted to ensure that the algorithm pauses until all pending operations are finished when running on the GPU [31].

To quantify the computational performance, the gigaflops are calculated using the best execution time on both the CPU and GPU and the formulas derived from the Linpack TPP benchmark of the HPC Challenge [26]. The computation follows the formulas in Equations 4.1 and 4.2. This allows a comparison of performance across various matrix sizes.

4.2.2 A\b on CPUs and GPUs with Specific Problem Types

The examination of the backslash operator [30] on CPUs and GPUs, focusing on specific problem types characterized by distinct matrix patterns, follows a framework akin to the analysis involving generated matrices in the preceding section, Chapter 4.2.1 [31]. Therefore, comparable steps are reiterated, with an emphasis on delineating the differences between the two analyses.

Similarly, any data in the GPU is first cleared and the GPU is initialized with the `gpuDevice` function [16], the available memory in the CPU and GPU is then determined in GB [31]. Rather than generating matrices, data pertinent to a specific problem type is loaded onto MATLAB. Upon loading, the dimensions of matrix A and vector b are determined, and a comparison is conducted between the size of matrix A and the available GPU memory, as the

former may exceed the latter. In such cases, matrix A and vector b are optimally divided into several sub-matrices and sub-vectors to fit within the available GPU memory at its largest dimension. Otherwise, an error may potentially arise if this condition is not met. It is essential to consider the indices of the elements so that all elements in matrix A and vector b are incorporated into the sub-matrices and sub-vectors, respectively, and they are computed only once to obtain the measured time. To ensure the precision of the time taken and facilitate a fair assessment, matrix A and vector b are also divided into sub-matrices and sub-vectors when running on the CPU.

The division of matrix A and vector b is a deliberate choice due to the limited support for the backslash operator from MATLAB PCT [30], in contrast to the fully supported `pcg` function [18] mentioned in Chapter 4.1.1. Consequently, the backslash operator struggles to handle a large problem in its entirety, even with MATLAB PCT, potentially facing errors linked to the limited memory of the mobile GPUs. Additionally, the partitioning of matrix A and vector b is based on their dimensions without the precision required to solve for the unknown x accurately. This approach aligns with the primary goal of assessing how the mobile GPUs perform when tasked with handling all sub-systems combined, as their total size matches that of the loaded problem—a large and sparse system. Therefore, methods like the block-Jacobi preconditioner [32, 33] are not employed to accurately partition matrix A and vector b and solve $x = A\backslash b$.

After configuring the sub-matrices, the test function, named `run`, is executed in a manner resembling the analysis of the backslash operator using generated matrices in the previous section [30, 31], outlined in Algorithm 11. In Algorithm 11, Lines 2 and 3, the functions `hpcCPU` and `hpcGPU` are integrated to calculate gigaflops using the formulas presented in Equations 4.1 and 4.2 [26] when the backslash operator is executed on the CPU and GPU, respectively. Algorithm 12 provides an overview of `hpcGPU`, demonstrating the key distinction between the two functions: `hpcGPU` accounts for the time taken to introduce overhead, which is then deducted from the execution time, as indicated in Lines 7 to 14 of Algorithm 12.

In the described functions, the backslash operator [30] is invoked multiple times to obtain the optimal execution time for the entire matrix A, as illustrated in Lines 3 to 6 of Algorithm 12. The backslash operator is employed when the functions `tSolveCPU` and `tSolveGPU` are called to execute on the CPU and GPU, respectively [31]. The flow of `tSolveGPU` is outlined in Algorithm 13.

In Algorithm 13, the execution times for each sub-matrix are consolidated, as depicted in Line 15 of Algorithm 13, yielding the total execution time for the entire matrix A. The `tSolve` functions retain a largely consistent structure for both the CPU and GPU, with the exception that on the GPU, data is transferred from the CPU to the GPU using `gpuArray` [21], introduced in the MATLAB PCT, as indicated in Lines 9 and 10 of Algorithm 13. Furthermore, a wait function is implemented to temporarily halt the program, allowing for the completion of all pending operations, as seen in Line 12 of Algorithm 13. Additionally, the GPU memory is cleared after each execution of the backslash between the sub-matrix and sub-vector, as illustrated in Line 14 of Algorithm 13, facilitating the smooth operation of the subsequent backslash operation between the next sub-matrix and sub-vector on the GPU [31].

Algorithm 11 Test function for A\b on CPUs and GPUs with Specific Problem Types

1: function $\mathrm{run}(\mathrm{A}, \mathrm{b})$
2: $[\mathrm{gflops}_{\mathrm{CPU}}, \mathrm{time}_{\mathrm{CPU}}] = \mathrm{hpcCPU}(\mathrm{A}, \mathrm{b})$
3: $[\mathrm{gflops}_{\mathrm{GPU}}, \mathrm{time}_{\mathrm{GPU}}] = \mathrm{hpcGPU}(\mathrm{A}, \mathrm{b}, @(\,)\ \mathrm{wait}_{\mathrm{GPU}}(\mathrm{gpu}))$
4: return $[\mathrm{gflops}_{\mathrm{CPU}}, \mathrm{time}_{\mathrm{CPU}}, \mathrm{gflops}_{\mathrm{GPU}}, \mathrm{time}_{\mathrm{GPU}}]$
5: end function

Algorithm 12 HPC function for A\b on GPUs with Specific Problem Types

```
function hpcGPU(A, b)
    Specify number of tests N_test and initialize t_test = ∞
    for iterations i = 0: N_test, do
        t = tSolveGPU(A, b, wait_GPU(gpu))
        t_test = min(t, t_test)
    end for
    Initialize t_overhead = ∞
    for iterations i = 0: N_test, do
        t = tic
        wait_GPU(gpu)
        t = toc
        t_overhead = min(t, t_overhead)
    end for
    t_GPU = t_test − t_overhead
    flops = 2/3 * n^3 + 3/2 * n^2
    gflops_GPU = flops/t_GPU/10^9
    return [gflops_GPU, t_GPU]
end function
```

Algorithm 13 Solve A\b function on GPUs with Specific Problem Types

1: function $\mathrm{tSolveGPU}(A, b, \mathrm{wait})$
2: Initialize time t_{total}
3: Calculate the number of sub-matrices, N_{parts}, where A needs to be divided
4: for iterations $j = 1: N_{parts}$
5: Compute start and end indices for the rows of the sub-matrix, A_{sub}
6: for iterations $i = 1: N_{parts}$
7: Compute start and end indices for the columns of A_{sub}
8: Copy the respective elements from A and b to A_{sub} and b_{sub}
9: $A_{sub} = \mathrm{gpuArray}(A)$
10: $b_{sub} = \mathrm{gpuArray}(b)$
11: $t_{sub} = \mathrm{tic}$
12: $\mathrm{wait}(\mathrm{gpu})$
13: $t_{sub} = \mathrm{toc}$
14: $\mathrm{reset}(\mathrm{gpu})$
15: $t_{total} = t_{total} + t_{sub}$
16: end for
17: end for
18: return $[t_{total}]$
19: end function

4.3 Data Handling Capability of GPUs

The evaluation of the GPU's data processing capability is conducted through three distinct sub-sections [34]: the transfer speed of data between CPUs and GPUs, read-write speed between CPUs and GPUs, and rate of computationally intensive operation on GPUs. This approach aims to quantify GPU performance, recognizing the substantial variations across different GPU devices. It provides valuable insights into the data or computation requirements for the GPU to outperform the CPU effectively, extending beyond the execution of the ICPCG method. The overall workflow is presented in Figure 4.1 [34].

In the first sub-section, the focus is on assessing how swiftly data can be sent to and read from the GPU. The speed of data transfer is intricately linked to the speed and activity level of the Peripheral Component Interconnect (PCI) bus, given that GPUs are integrated into the PCI bus. Additionally, the measurements in this test encompass some overheads, mirroring real-world GPU applications [34].

The procedure begins by initializing the GPU with `gpuDevice` [16] and declaring a double-precision array of data sizes in bytes, ranging from 2^{14} to 2^{18}. Two vectors are then generated, with dimensions corresponding to the array of data sizes, one on the GPU and the other on the CPU. Subsequently, memory is allocated, and the data on the CPU is transmitted to the GPU using the `gpuArray` function [21]. Following this, the data on the GPU is transferred back to the host memory using the `gather` function [35]. To accurately measure the time taken during the data transfer, the `gputimeit` function [22] is employed instead of the regular `timeit` function [36]. `gputimeit` ensures that all GPU operations are completed before recording the time and compensates for the overhead time. All the mentioned functions, except `timeit`, belong to the MATLAB PCT. Utilizing the timings, the send and gather bandwidths are calculated in GB.

The second sub-section evaluates the read-write speed between CPUs and GPUs by executing memory-intensive operations. The objective is not to assess computational speed but rather to evaluate the efficiency of memory read and write operations for each floating-point operation. Given that many operations involve minimal computation per array element, they are predominantly influenced by the time required to fetch or write data. To assess this, the `plus` function, with straightforward computation, is implemented. This function performs one memory

read and one memory write for each floating-point operation, making it a reliable indicator of the read-write operation speed and it should be limited by memory access speed [34].

Using the double-precision array of data sizes in bytes, vectors with varying dimensions are generated on both the CPU and GPU. The `plus` function is then applied to the vectors on the respective platform. This function has a computational density of $1/2$ flops per element. To measure the time on the GPU, the `gputimeit` function [22] is employed, whereas the `timeit` function [36] is used for the CPU measurements. Once the timings are obtained for each size and hardware, the read-write bandwidth is computed in GB [34].

In the ultimate sub-section, the focus shifts to testing the rate of operations with high computational intensity, where the number of floating-point computations executed per element read from and written to memory is substantial. In such scenarios, the memory speed becomes less critical, and the limiting factor is the number and speed of floating-point units, given the operations' high computational density. To examine this, the matrix-matrix multiplication is chosen as a computationally intensive operation. The total number of floating-point calculations is given by $\text{flops}(N) = 2N^3 - N^2$, where N denotes the size of the matrix [34].

The process initiates by expanding the range of the existing double-precision array of data-sizes in bytes, now spanning from 2^{12} to 2^{24}. Subsequently, two input square matrices, A and B, are generated for the multiplication operation $A * B$. This matrix-matrix multiplication is executed on both the CPU and GPU. Similarly, on the GPU, the `gpuArray` function [21] is utilized, and the `timeit` and `gputimeit` functions measure the time taken on the CPU and GPU [22, 36], respectively. The outcome is a matrix written to the corresponding platform. The timing data is then utilized to calculate the rate of operations in gigaflops. In total, the number of elements read or written is $3N^2$, with a computational density of $(2N - 1)/3$ flops per element, marking a higher level of computational intensity compared to the previous sub-section [34].

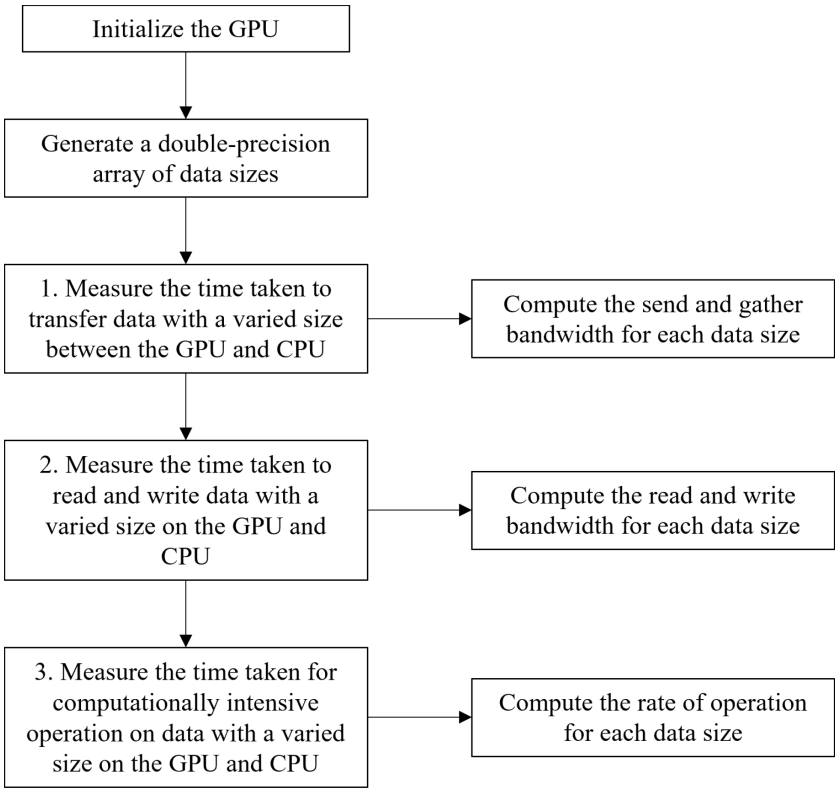

Figure 4.1: Flowchart of Data Handling Capability of GPUs.

4.4 Resource Contention on CPUs using Single Program Multiple Data Statements

This section is dedicated to evaluating resource contention on CPUs, with a specific focus on understanding how the number of concurrent processes and data size influence the speedup in various operations, including the execution of the ICPCG method covered in Chapter 4.1.2. In this sub-chapter, the operations encompass summation, discrete Fast Fourier Transform (DFFT), and matrix-matrix multiplication, and their examination aids in demonstrating the significance of resource contention for memory access [37].

To facilitate these assessments on CPUs, the `parpool` function [28] is employed to create a parallel pool of workers or processes within a process-based environment [29]. As described in the earlier methodology on implementing the ICPCG method using SPMD statements (Chapter 4.1.2), `parpool` would get MATLAB to establish a pool on the local machine, assigning one

worker to each physical CPU core. These parallel workers execute computational tasks using SPMD statements, enabling the parallelized code to run on workers within the same multi-core CPU. This allows each worker to work on a different data set or portion of the distributed data while communicating with other parallel workers during parallel computations [27, 28]. Once the parallel workers are configured, a matrix is generated, providing the foundation for the subsequent operations [37].

The first part of this evaluation explores the impact of the number of concurrent processes on the speedup, and this is achieved through the execution of functions that are summation, DFFT, and matrix-matrix multiplication. These tests employ either a fixed-size vector or a fixed-size square matrix with the same total elements, and the number of parallel processes varies, ranging from one to the total count of available parallel processes. Each function is executed multiple times to obtain an average reading for accurate timings. Additionally, a timing function is created to run the computation functions numerous times using the SPMD statements, retaining the minimum execution time observed for each level of concurrency [37].

Conversely, to assess the influence of data size on the speedup, speed tests are conducted on a vector or a square matrix of various dimensions, where the total number of elements between a vector and a square matrix remains the same. This part of the evaluation encompasses additional functions such as LU decomposition, singular value decomposition (SVD), and eigenvalue computation. These additional functions and varying data size help investigate the effects of different memory access patterns and the impacts of different data sizes. In this scenario, the tests are performed using either a single or all available parallel processes. Similarly, a timing function is used to run the computation functions numerous times with the SPMD statements, storing the fastest execution time for the given level of concurrency [37]. Figure 4.2 shows the flow of the resource contention evaluation.

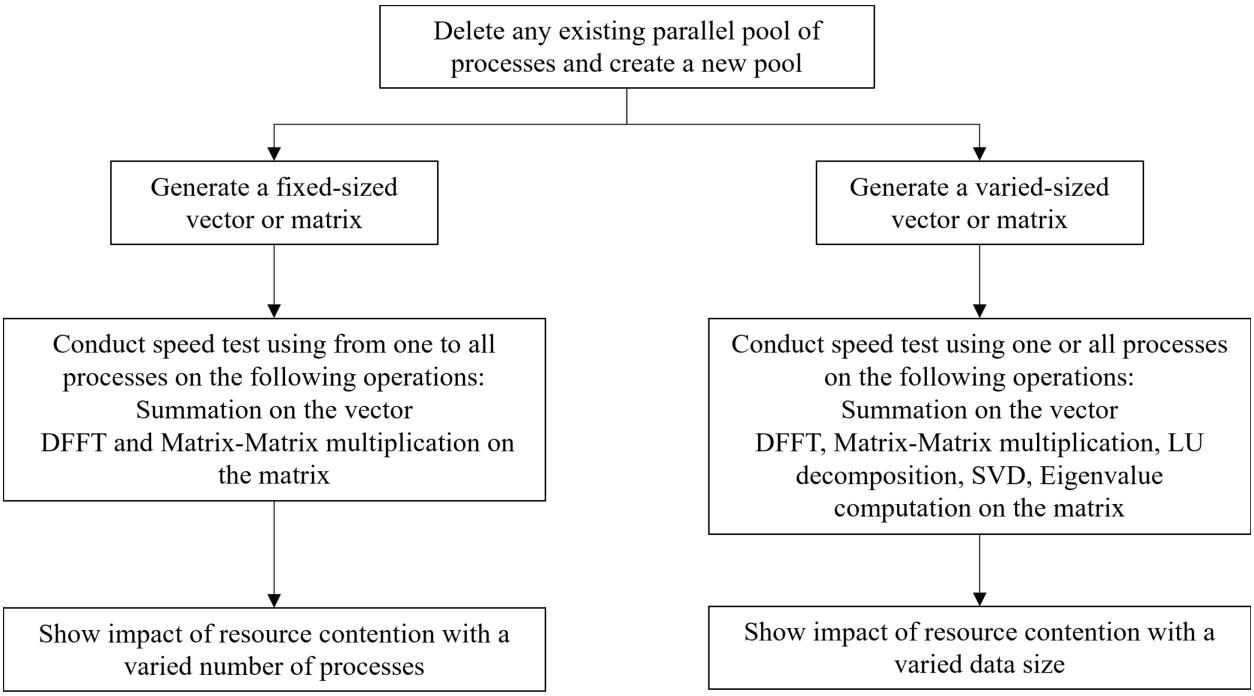

Figure 4.2: Flowchart of Resource Contention Evaluation.

4.5 MATLAB's GPUBench

The final inclusion in the evaluations is GPUBench, developed by the MathWorks PCT Team. GPUBench is a utility that measures the timing of various MATLAB GPU tasks and provides an estimate of the GPU's peak performance in flops. It generates a comprehensive HyperText Markup Language (HTML) report, illustrating the GPU's performance relative to the pre-existing performance data from various other GPUs. It is specifically crafted for comparing GPU hardware and does not assess GPU performance variations across different MATLAB release. However, it is also possible to implement the tests on the CPU to evaluate its performance [38].

In GPUBench, the initialization process involves setting up the data object, CPU, and GPU. Prior to the execution of each task, GPUBench determines the maximum allowable data size in either single- or double-precision, based on the available memory on the respective platform. Subsequently, it performs tasks such as matrix-matrix multiplication, the backslash operator, DFFT using the generated data on both the CPU and GPU, considering both single- and double-precision data types. To prevent program crashes, a safety factor variable is incorporated, restraining the

amount of required memory for the generated data. It is anticipated that matrix-matrix multiplication and the backslash operator involve regular memory access, while DFFT entails irregular memory access. Each task is executed for a range of array sizes. The outcomes are then presented in an HTML report [38].

Chapter 5
Results and Discussion

In this chapter, we present the outcomes derived from the comprehensive analyses and evaluations delineated in the previous chapter. All methodologies expounded upon in Chapter 4 underwent testing on two sets of CPU and GPU configurations, which are found in mobile devices such as laptops. The first pair features the 10th Generation Intel® Core™ i7 processor, i7-10510U, coupled with the NVIDIA GeForce GTX 1650 with Max-Q Design. The second pair involves the 4th Generation Intel® Core™ i7 processor, i7-4710HQ, paired with the NVIDIA GeForce GTX 970M. The Turing architecture is incorporated in the GeForce GTX 1650 with Max-Q Design, whereas the GeForce GTX 970M is based on the Maxwell 2.0 design [39, 40]. For an in-depth specification of each CPU and GPU, kindly refer to the Appendix.

As discussed in Chapter 4, additional methods beyond the ICPCG approach are employed to assess the performance of the mobile GPUs. Consequently, the outcomes derived from these supplementary methods corroborate the results obtained through the ICPCG method, confirming the accurate implementation of the ICPCG method and providing insights into how mobile GPUs operate, particularly in the context of executing iterative solvers.

The specific problem types used for conducting the analyses are thermal and electromagnetics problems. The thermal problem, thermal1, chosen for evaluation exhibits a distinctive pattern, as depicted in Figure 5.1, with a structural rank of 82,654. It is characterized as real and SPD [41]. On the other hand, the electromagnetics problem, 2cubes_sphere, presents a different unique pattern, as illustrated in Figure 5.2, with a structural rank of 101,492. It is also real and SPD [42].

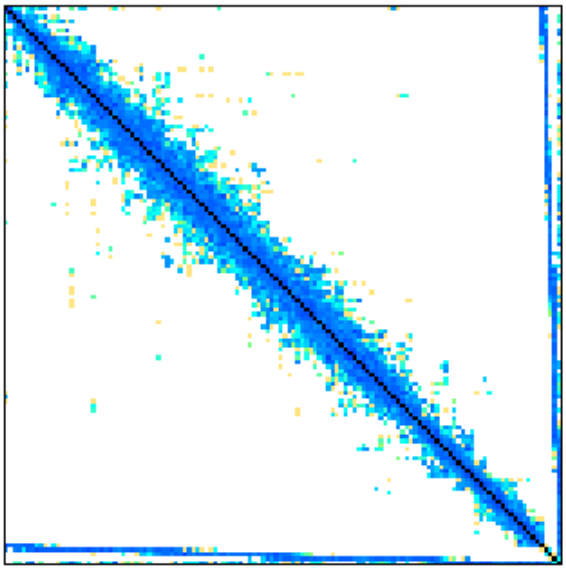

Figure 5.1: Thermal problem pattern (thermal1) [41].

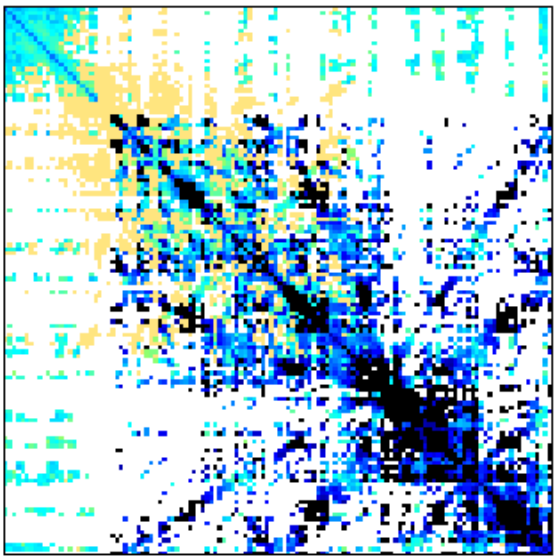

Figure 5.2: Electromagnetics problem pattern (2cubes_sphere) [42].

5.1 ICPCG on CPUs and GPUs with Specific Problem Types

In this segment, the assessment of the ICPCG method is carried out on the specified problem types, namely thermal1 and 2cubes_sphere [41, 42], utilizing the two designated sets of CPU and GPU configurations. It is important to reiterate the key distinction between the `pcg` function and the backslash operator, as mentioned in Chapter 4.1.1. The `pcg` function has the full support from MATLAB PCT, unlike the backslash operator [18, 30]. Hence, there is no need to partition matrix A of the loaded problem in this part of the evaluation.

5.1.1 ICPCG using Parallel Computing Toolbox Commands

The results presented in Tables 5.1 and 5.2 are derived from the parallel application of the ICPCG method using MATLAB PCT functions, as outlined in Algorithm 8. The GPU execution timings are measured through both the `tic` and `toc` functions and the `gputimeit` function [19, 20, 22], whereas the CPU execution timings are obtained exclusively using the `tic` and `toc` functions.

Table 5.1: Time taken and Gigaflops for ICPCG (thermal1).

	i7-10510U	**GTX 1650 Max-Q**	**i7-4710HQ**	**GTX 970M**
Time taken for ichol (s)	0.012722	-	0.012761	-
Time taken for pcg using tic-toc (s)	413.928124	860.500157	614.373076	1271.697566
Time taken for pcg using gputimeit (s)	-	887.115585	-	1269.049949
Gigaflops from tic-toc timing	909.467649	437.483056	612.745338	296.024973
Gigaflops from gputimeit timing	-	424.357597	-	296.642570

Table 5.2: Time taken and Gigaflops for ICPCG (2cubes_sphere).

	i7-10510U	**GTX 1650 Max-Q**	**i7-4710HQ**	**GTX 970M**
Time taken for ichol (s)	0.031839	-	0.039136	-
Time taken for pcg using tic-toc (s)	490.549099	2208.567411	775.176322	4088.910038
Time taken for pcg using gputimeit (s)	-	2212.471140	-	4089.244489
Gigaflops from tic-toc timing	1420.794670	315.575400	899.110983	170.453627
Gigaflops from gputimeit timing	-	315.018593	-	170.439685

It is noteworthy that the `ichol` operation [17], being less computationally intensive, demonstrates minimal performance variation between different CPU generations, with a marginal difference of 0.039ms for the smaller problem size, thermal1, and a much larger discrepancy of 7.3 ms for the larger data size, 2cubes_sphere. Conversely, the `pcg` operation [18], being significantly more computationally intensive, exhibits substantial differences across all CPUs and GPUs.

Upon multiple executions, the GeForce GTX 1650 with Max-Q Design exhibits better performance using `tic` and `toc` (without a function handle) compared to `gputimeit` (with a function handle). In contrast, the GeForce GTX 970M demonstrates similar performance using both methods.

In the initial CPU-GPU configuration (Core i7-10510U and GeForce GTX 1650 with Max-Q Design), the `pcg` operation on thermal1 using the GeForce GTX 1650 with Max-Q Design yields a speed up of 0.481 compared to the Core i7-10510U, indicating that the GPU is 107.89% slower than the CPU. For the `pcg` operation on 2cubes_sphere, the speedup is 0.222, signifying that the GPU is 350.22% slower than the CPU.

In the second CPU-GPU configuration (Core i7-4710HQ and GeForce GTX 970M), the `pcg` operation on thermal1 using the GeForce GTX 970M results in a speedup of 0.483 compared to the Core i7-4710HQ, indicating that the GPU is 106.99% slower than the CPU. For the `pcg` operation on 2cubes_sphere, the speedup is 0.19, signifying that the GPU is 427.48% slower than the CPU.

In both CPU-GPU configurations, it is observed that the CPUs have surpassed the GPUs in performance when tackling the specified problem types. This trend becomes particularly noticeable when GPUs, specifically mobile graphics chips, are tasked with handling double-precision variables, a pattern consistently seen in the subsequent sub-chapters. Therefore, these GPUs demonstrate inefficiency in handling intensive computations, such as executing iterative sparse solvers, primarily attributed to their limited memory bandwidth [43]. Additionally, mobile devices, including laptops and tablets, have strict power and thermal constraints. Mobile GPUs are designed to operate within these constraints, which can limit their performance compared to desktop GPUs [43].

5.1.2 ICPCG using Single Program Multiple Data Statements

In this segment, the ICPCG method is implemented on CPUs utilizing the parallel pool from the concurrent execution of SPMD statements [27]. Given that both tested CPUs, Core i7-10510U and Core i7-4710HQ, feature four cores [44, 45], the number of parallel workers ranges from one to four. Moreover, the outcomes of applying the ICPCG to thermal1 are detailed in Tables 5.3 and 5.4, while those for 2cubes_sphere are presented in Tables 5.5 and 5.6. These results serve as the basis for generating Figures 5.3 and 5.4 in the case of thermal1, and Figures 5.5 and 5.6 for 2cubes_sphere.

Referring to Figures 5.3 and 5.4, the results indicate that both the Core i7-10510U and the Core i7-4710HQ exhibit performance degradation with an increasing number of parallel workers. The Core i7-10510U experiences a total time increase of 76.16% from one to four parallel workers, averaging a 20% increment for each additional worker. Additionally, the Core i7-4710HQ shows a total time increase of 50.2% from one to four parallel workers, with an average increment of 14.56% for each additional worker.

Furthermore, considering Figures 5.5 and 5.6, the findings reveal that both CPUs manifest performance degradation with a rising number of parallel workers. The Core i7-10510U registers a total time increase of 121.94% when employing from one to four parallel workers, averaging a 30.55% increment for each additional worker. Similarly, the Core i7-4710HQ demonstrates a total time increase of 102.78% from one to four parallel workers, with an average increment of 26.63% for each additional worker. It is apparent that the decline in performance becomes more pronounced with an increase in problem size.

Upon analyzing the results presented in this section and those in the subsequent sub-chapter (Chapter 5.4), it is evident that resource contention occurs when implementing the ICPCG method on the CPUs. This contention becomes more prominent with an expansion in problem size [37].

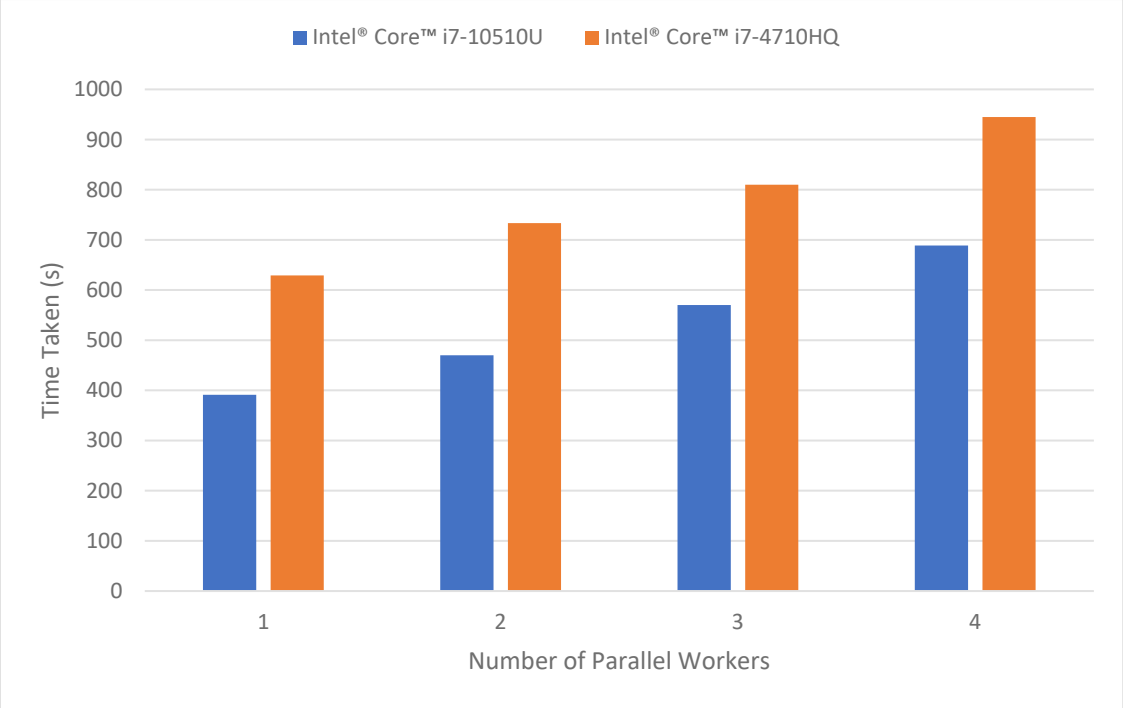

Figure 5.3: ICPCG Execution Time vs Number of Parallel Workers (thermal1).

Table 5.3: Execution Times for ICPCG (thermal1).

Number of Parallel Workers	**Core i7-10510U Time taken (s)**	**Core i7-4710HQ Time taken (s)**
1	391.064867	629.175917
2	470.019997	733.507735
3	570.080991	809.8314
4	688.886328	944.9925

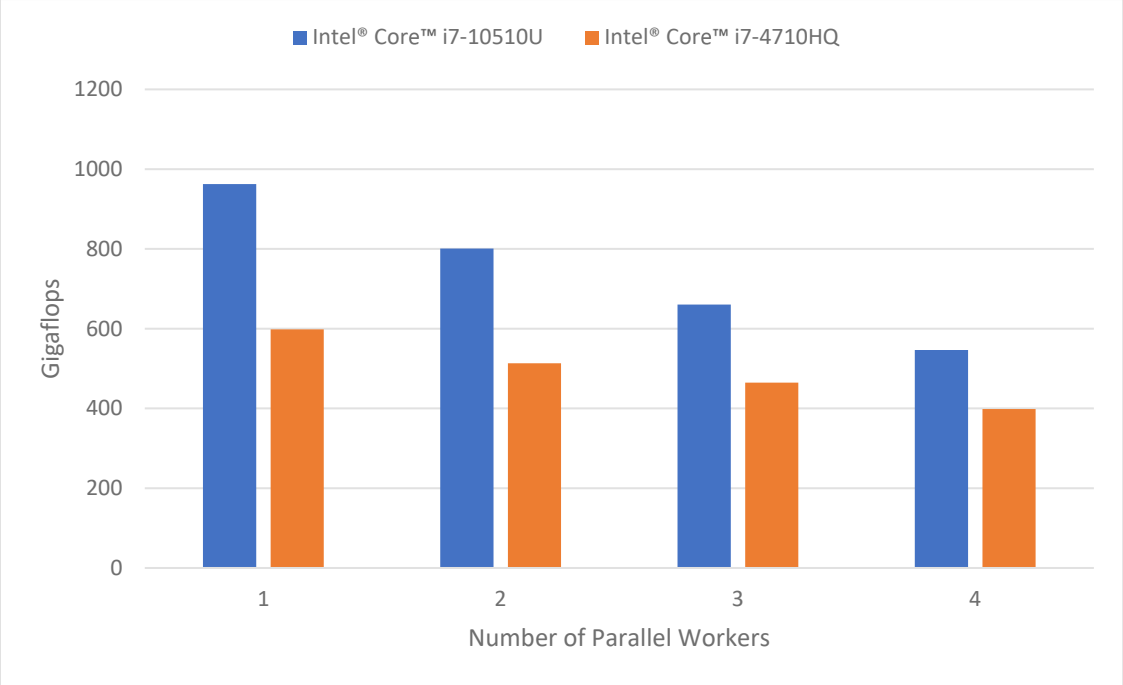

Figure 5.4: Gigaflops vs Number of Parallel Workers (thermal1).

Table 5.4: Gigaflops for ICPCG (thermal1).

Number of Parallel Workers	**Core i7-10510U Gigaflops**	**Core i7-4710HQ Gigaflops**
1	962.63886	598.329065
2	800.932386	513.224633
3	660.352202	464.855077
4	546.467861	398.367435

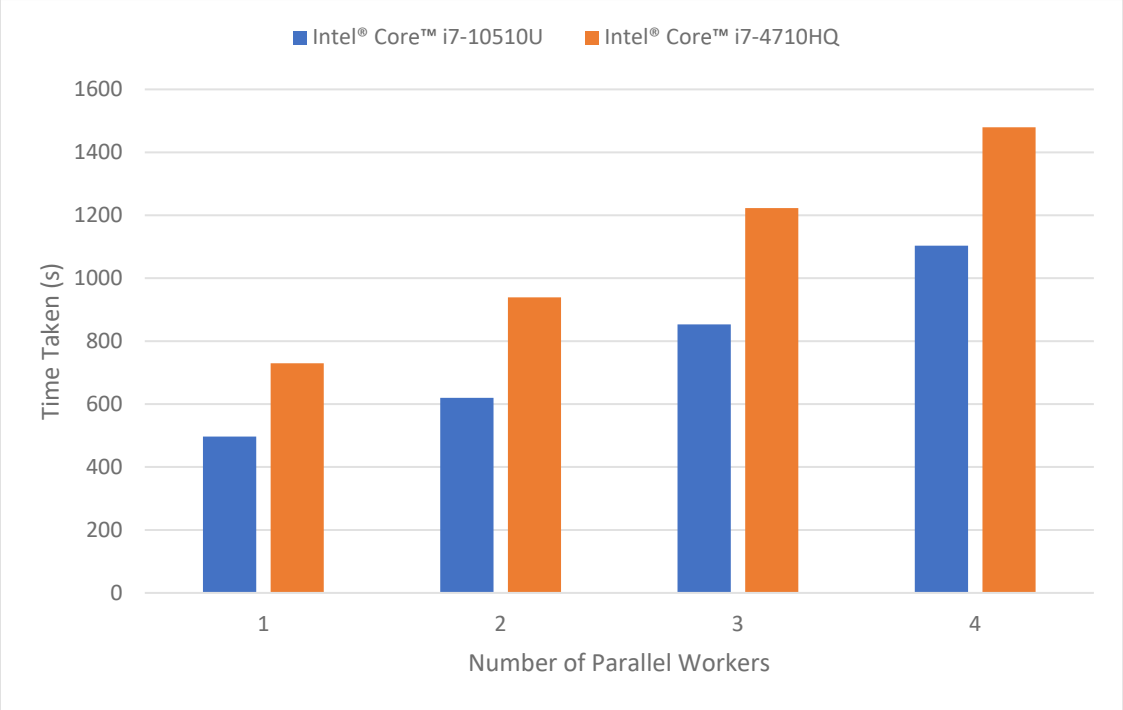

Figure 5.5: ICPCG Execution Time vs Number of Parallel Workers (2cubes_sphere).

Table 5.5: Execution Times for ICPCG (2cubes_sphere).

Number of Parallel Workers	**Core i7-10510U Time taken (s)**	**Core i7-4710HQ Time taken (s)**
1	497.054921	729.7716
2	620.072275	939.167759
3	853.181401	1222.934716
4	1103.156395	1479.805379

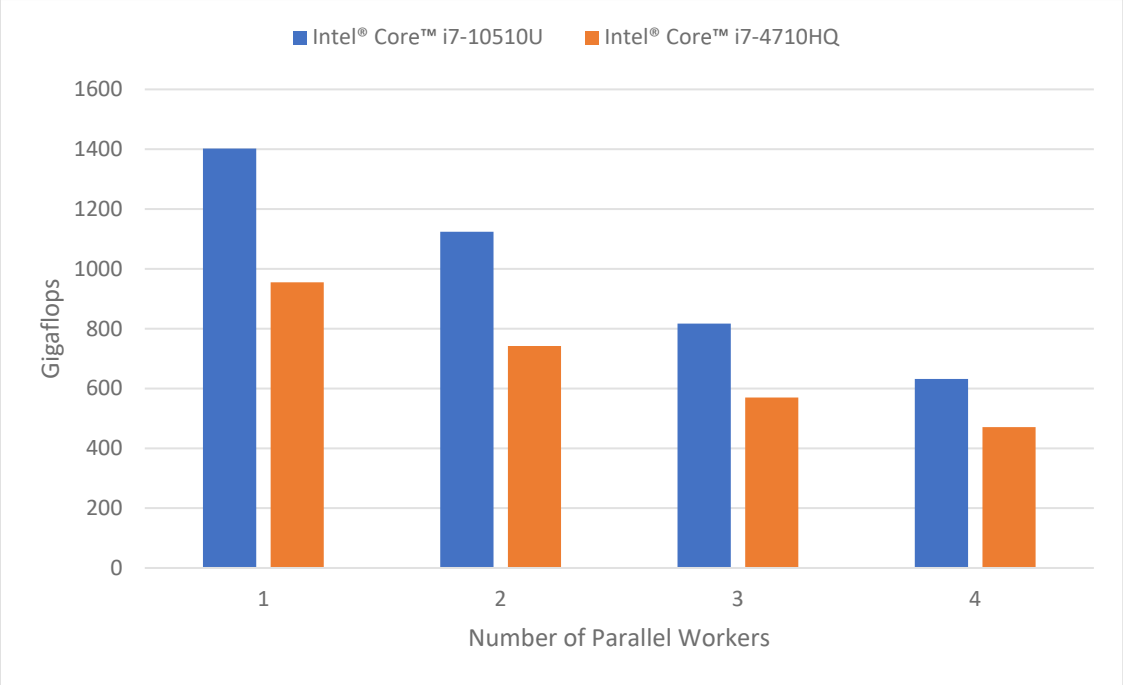

Figure 5.6: Gigaflops vs Number of Parallel Workers (2cubes_sphere).

Table 5.6: Gigaflops for ICPCG (2cubes_sphere).

Number of Parallel Workers	**Core i7-10510U Gigaflops**	**Core i7-4710HQ Gigaflops**
1	1402.198259	955.051614
2	1124.013397	742.114002
3	816.906632	569.915577
4	631.795771	470.987303

5.2 Backslash on CPUs and GPUs

The backslash operator [30] underwent testing on generated matrices [31] and the two specific problem, thermal1 and 2cubes_sphere [41, 42], where each specific problem showcasing distinct patterns illustrated in Figures 5.1 and 5.2, respectively. As highlighted in Chapters 4.1.1 and 5.1, the backslash operator is not fully supported by MATLAB PCT, unlike the `pcg` function [18, 30]. Consequently, it may encounter potential errors related to limited memory when operating on the mobile GPUs. Therefore, the backslash operator is implemented on matrices sized according to the maximum available memory of the mobile GPUs. The outcomes acquired in this evaluation phase align with the results derived from the ICPCG method in the preceding sub-chapter (Chapter 5.1).

5.2.1 A\b on CPUs and GPUs with Generated Matrices

The generated matrices encompass two data types: single- and double-precision. In MATLAB, single-precision variables are stored as 4-byte (32-bit) floating-point values, while double-precision variables are stored as 8-byte (64-bit) floating-point values [46]. Consequently, the expected range of the generated matrices for the single-precision data type exceeds that of the double-precision data type.

In the first CPU-GPU setup featuring the Core i7-10510U and the GeForce GTX 1650 with Max-Q Design, matrices of the single-precision class cover a range of nine sizes, starting from 1024×1024 and extending up to 17408×17408. Conversely, in the second CPU-GPU configuration with the Core i7-4710HQ and the GeForce GTX 970M, single-precision class matrices are available in eight different sizes, ranging from 1024×1024 to 15360×15360. These ranges are also reflected in Table 5.7. Figures 5.7 and 5.8 are generated based on the gigaflops data presented in Table 5.7.

In Figure 5.7, the performance of the GeForce GTX 1650 with Max-Q Design significantly outpaces that of the Core i7-10510U, reaching a peak speedup of 3.374 when comparing the GPU to the CPU, as illustrated in Figure 5.11. Similarly, Figure 5.8 shows that the performance of the GeForce GTX 970M surpasses that of the Core i7-4710HQ, achieving a peak speedup of 4.678 when comparing the GPU to the CPU, as depicted in Figure 5.12. However, in cases where the

generated matrix size is notably small, both CPUs, Core i7-10510U and Core i7-4710HQ, exhibit better performance than the GPUs, GeForce GTX 1650 with Max-Q Design and GeForce GTX 970M, respectively. Therefore, at the smallest matrix size of 1024×1024, the speedup of the GeForce GTX 1650 with Max-Q Design compared to the Core i7-10510U is 0.746, as evident in Figure 5.11. Additionally, the speedup of the GeForce GTX 970M compared to the Core i7-4710HQ is 0.409, as observed in Figure 5.12.

Table 5.7: Gigaflops for A\b on Single-precision Matrix.

Matrix Size	**i7-10510U**	**GTX 1650 Max-Q**	**i7-4710HQ**	**GTX 970M**
1024×1024	86.987189	64.907872	62.81583	25.713381
3072×3072	188.748285	363.987757	127.163609	352.235054
5120×5120	234.061998	579.233822	155.228283	492.134057
7168×7168	262.873849	739.450249	167.426259	627.027793
9216×9216	274.043537	831.389600	177.324724	738.218882
11264×11264	275.591083	898.258581	177.094533	788.500995
13312×13312	282.730587	953.903685	196.568511	919.550602
15360×15360	280.948109	917.117924	228.412318	937.243363
17408×17408	283.465329	924.551816	-	-

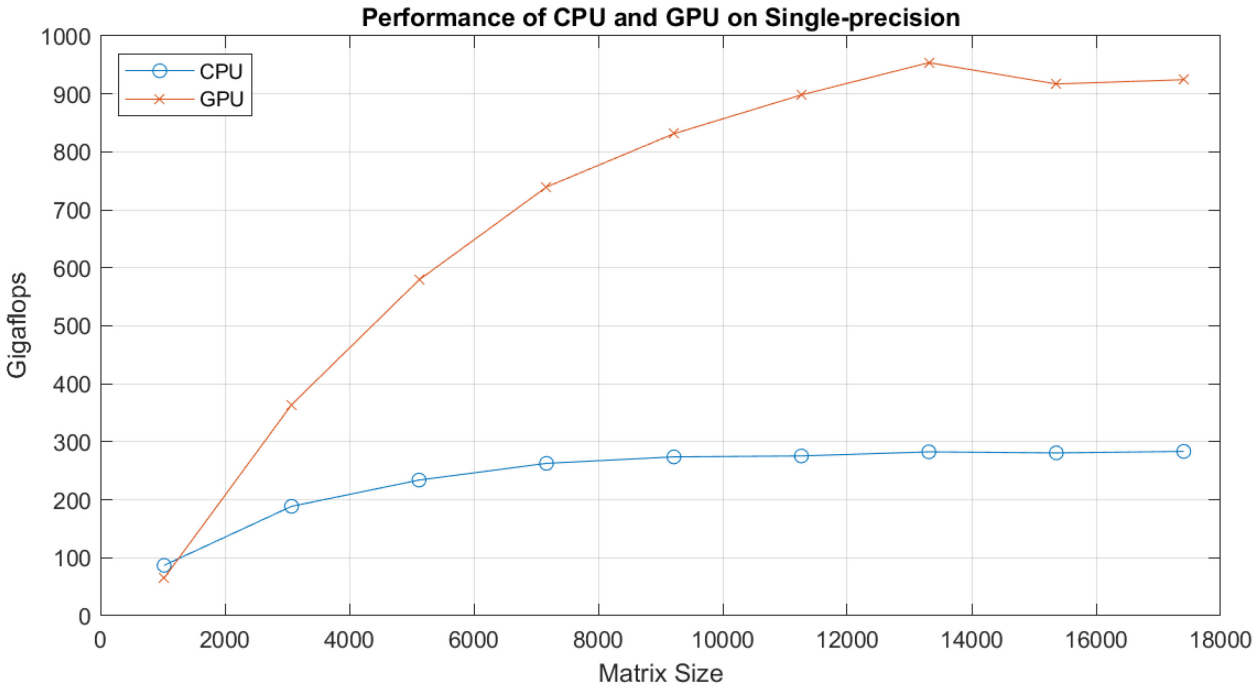

Figure 5.7: Performance of i7-10510U and GTX 1650 with Max-Q on Single-precision.

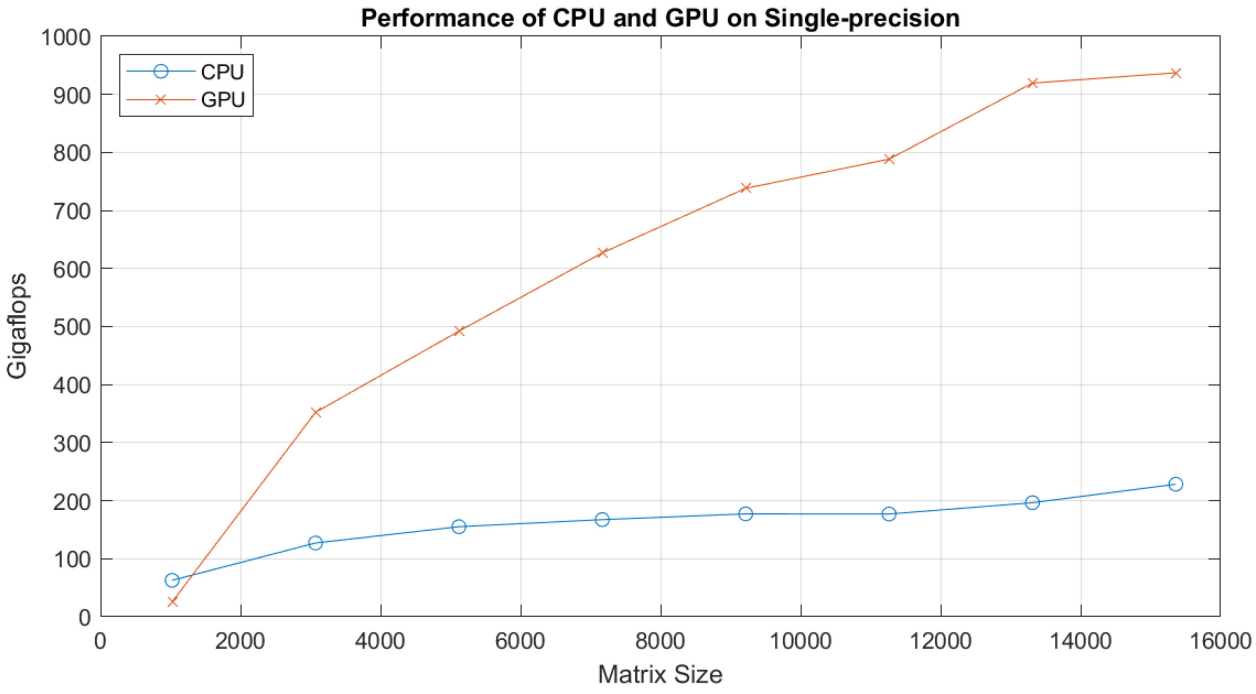

Figure 5.8: Performance of i7-4710HQ and GTX 970M on Single-precision.

Due to the increased space requirements of double-precision variables, the generated matrices of the double-precision class exhibit a more limited range. Both CPU-GPU configurations feature six matrix sizes ranging from 1024×1024 to 11264×11264, as detailed in Table 5.8. The gigaflops data presented in Table 5.8 serves as the basis for generating Figures 5.9 and 5.10.

In Figure 5.9, it is apparent that the performance of the GeForce GTX 1650 with Max-Q Design is inferior to that of the Core i7-10510U across all six matrix sizes, yielding an average speedup of 0.836 when comparing the GPU to the CPU, as indicated in Figure 5.11.

Meanwhile, Figure 5.10 illustrates that the performance of the GeForce GTX 970M is also subpar compared to the Core i7-4710HQ for all matrix sizes except the smallest. The GPU's performance appears to peak in gigaflops when the matrix size reaches 7168×7168. At the smallest matrix size of 1024×1024, the GPU exhibits a positive speedup of 1.102 compared to the CPU, as highlighted in Figure 5.12. However, as the matrix size increases, the speedup of the GPU compared to the CPU progressively decreases, reaching its minimum at 0.639 for the largest matrix size of 11264×11264.

Analyzing the outcomes, it is clear that the GPUs outshine the CPUs in performance particularly with single-precision matrices, which demand lower memory capacity. This superiority diminishes when dealing with double-precision matrices due to limited memory bandwidth on the mobile GPUs [43], as observed in the previous sub-chapter (Chapter 5.1.1). Similarly, the mobile GPUs are expected to operate within the strict power and thermal constraints [43]. Additionally, the GPUs exhibit a substantial advantage in parallizing the code for large data sizes, where the benefits of parallelization outweight the associated overhead. However, at smaller data sizes, the overhead involved in initiating and handling parallel tasks on the GPU becomes more pronounced, rendering the CPUs to be more efficient. As the data size expands, this overhead impact diminishes [13].

Table 5.8: Gigaflops for A\b on Double-precision Matrix.

Matrix Size	i7-10510U	GTX 1650 Max-Q	i7-4710HQ	GTX 970M
1024 × 1024	39.354915	33.259808	26.556333	29.251576
3072 × 3072	92.573312	81.085831	64.643107	58.813968
5120 × 5120	110.723287	92.779758	83.689847	65.260637
7168 × 7168	119.582394	97.292597	92.279303	68.117172
9216 × 9216	124.249193	100.911928	100.442205	67.929555
11264 × 11264	123.220378	102.693245	105.339049	67.288804

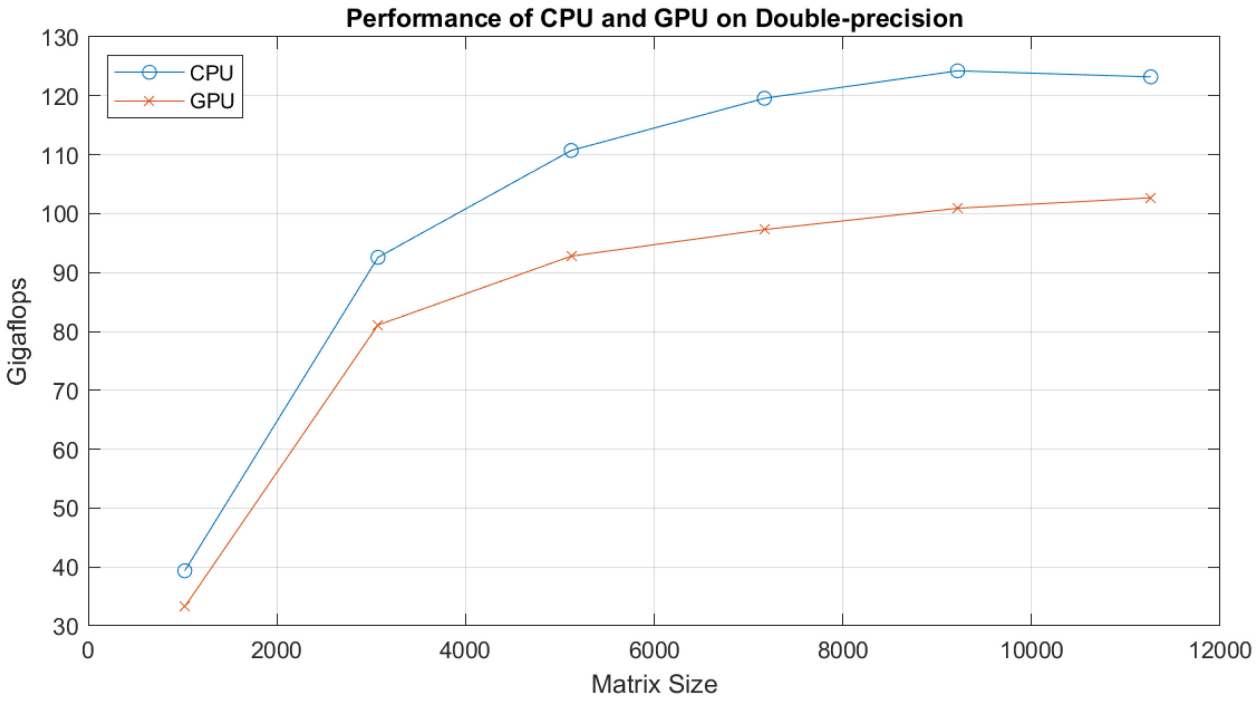

Figure 5.9: Performance of i7-10510U and GTX 1650 with Max-Q on Double-precision.

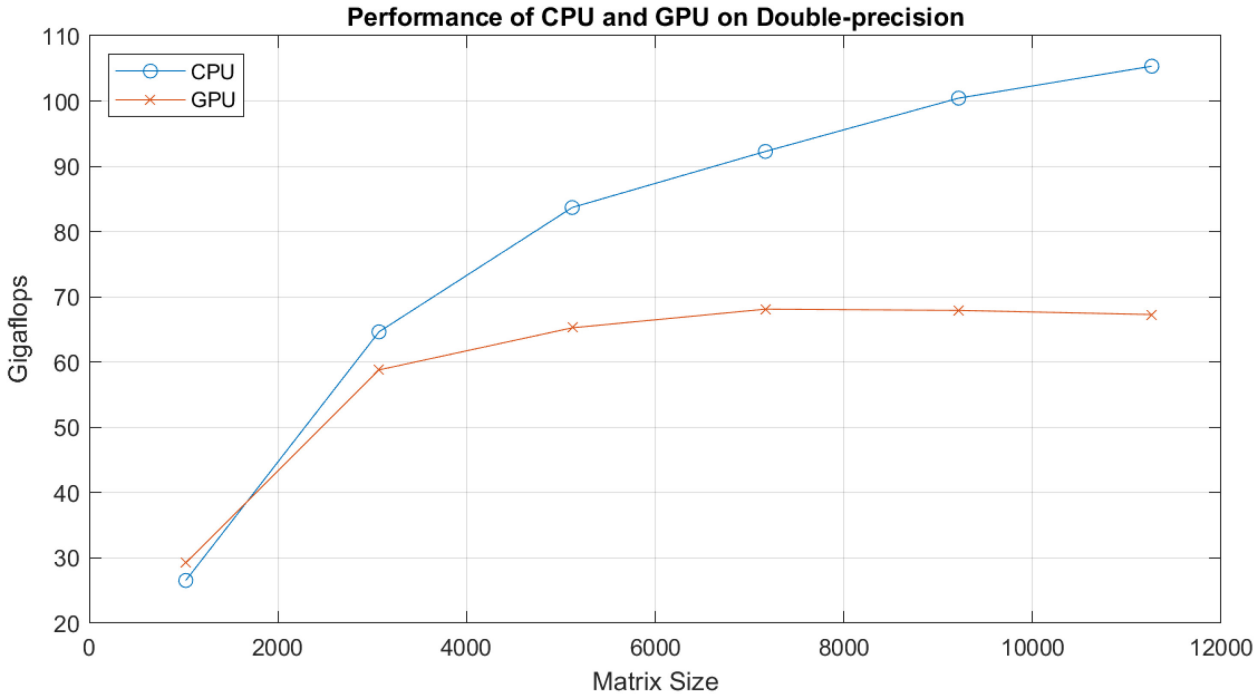

Figure 5.10: Performance of i7-4710HQ and GTX 970M on Double-precision.

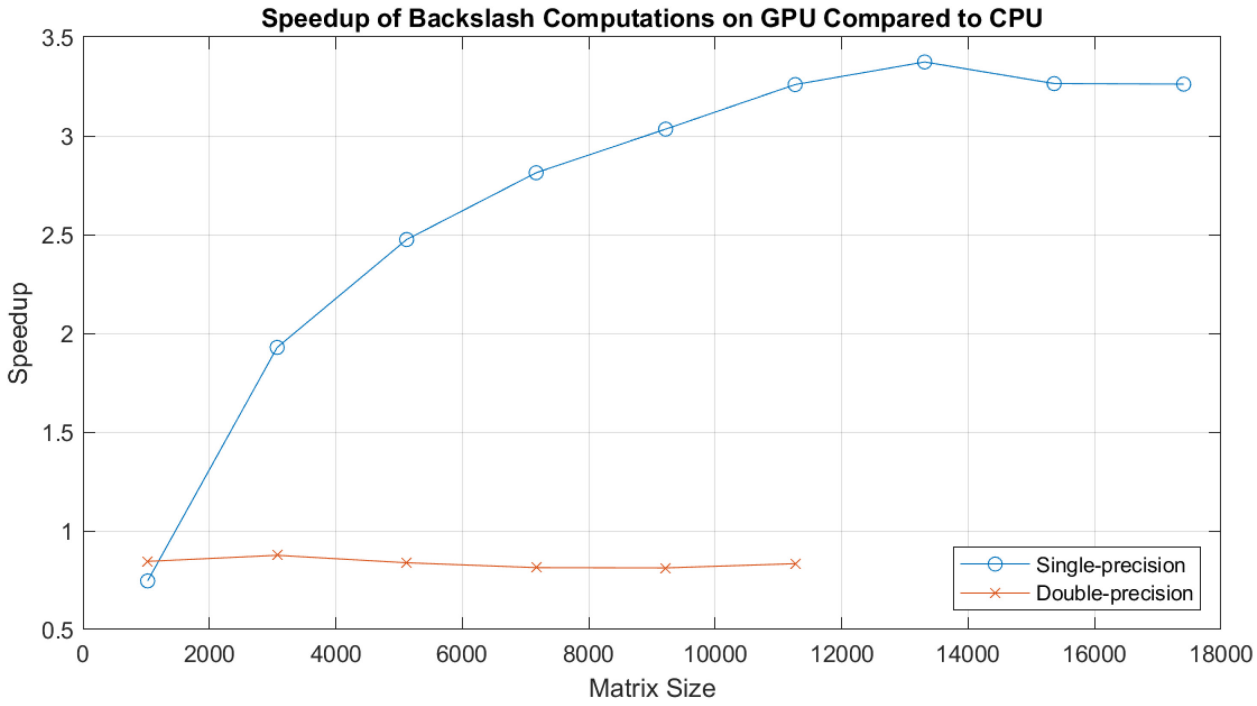

Figure 5.11: Speedup of Backslash on GTX 1650 with Max-Q Compared to i7-10510U.

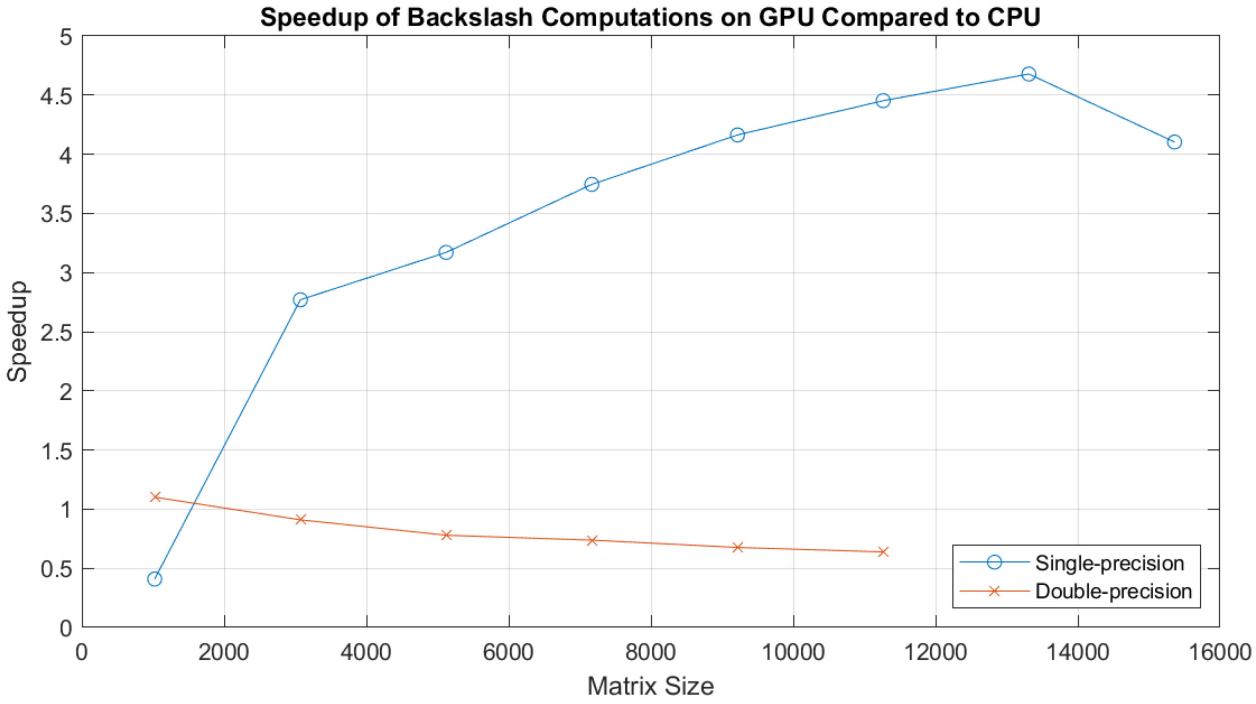

Figure 5.12: Speedup of Backslash on GTX 970M Compared to i7-4710HQ.

5.2.2 A\b on CPUs and GPUs with Specific Problem Types

Both problem types, thermal1 and 2cubes_sphere [41, 42], involve a double-precision data type, where elements are stored as 8-byte (64-bit) floating-point values in MATLAB [46]. To prevent memory overflow on both CPUs and mobile GPUs, matrix A of the loaded problem must be partitioned based on the platform's available memory, as matrix A requires the most extensive memory capacity.

As detailed in the methodology (Chapter 4.2.2), matrix A of the loaded problem is partitioned into sub-matrices, while the RHS vector b is divided into sub-vectors with a comparable dimension. The combined size of the sub-matrices equals that of the loaded problem as a whole. Given the emphasis on assessing the performance of the mobile GPUs with an equivalently large and sparse system of linear equations, rather than implementing methods such as the block-Jacobi preconditioner [32, 33] to precisely partition and solve the system, the partitioning of the system does not account for the accuracy of solving for the unknown x.

The GeForce GTX 1650 with Max-Q Design boasts a maximum available memory of 3306654107 bytes, equivalent to a double-precision matrix size of 20331×20331. Consequently, the matrices A for thermal1 and 2cubes_sphere surpass the GPU's maximum available memory. To circumvent this limitation, matrices A for thermal1 and 2cubes_sphere are divided into 25 sub-matrices each. This division facilitates the backslash operation to function on individual sub-matrices on the GPU without encountering potential errors. On the Core i7-10510U, matrix A is also partitioned into 25 sub-matrices when executing the backslash operator.

Similarly, the GeForce GTX 970M offers a maximum available memory of 2544900507 bytes, corresponding to a double-precision matrix size of 17836×17836. Analogously, matrices A for thermal1 and 2cubes_sphere exceed the GPU's maximum available memory. Leveraging the available memory of the GeForce GTX 970M, matrix A for thermal1 is segmented into 25 sub-matrices, while matrix A for 2cubes_sphere is divided into 36 sub-matrices. On the Core i7-4710HQ, matrix A is also segmented into 25 sub-matrices for thermal1 and 36 sub-matrices for 2cubes_sphere.

Tables 5.9 and 5.10 show the results for A\b on thermal1 and 2cubes_sphere, respectively, on both the CPU-GPU configurations. In the first CPU-GPU configuration (Core i7-10510U and GeForce GTX 1650 with Max-Q Design), the GPU performs poorly with a gigaflops speedup of 0.005 for thermal1 and 0.001 for 2cubes_sphere as compared to the CPU. For the second GPU-CPU configuration (Core i7-4710HQ and GeForce GTX 970M), the GPU also performs poorly with a gigaflops speedup of 0.002 for thermal1 and 0.003 for 2cubes_sphere.

The pattern identified in the earlier sub-chapters (Chapters 5.1.1 and 5.2.1) persists, showcasing the inefficiency of the mobile GPUs in managing demanding computations when tasked with double-precision data types. This stands in contrast to the commendable performance when dealing with single-precision data types, as emphasized in Chapter 5.2.1. Therefore, this consistent trend supports the notion that constraints such as power and thermal limitations, coupled with limited memory bandwidth, pose challenges for the mobile GPUs to deliver an optimal performance [43].

Table 5.9: Results for Backslash Operator (thermal1).

	i7-10510U	**GTX 1650 Max-Q**	**i7-4710HQ**	**GTX 970M**
Time taken for A\b (s)	0.575508	121.528429	1.165408	529.508559
Gigaflops	654124.54916	3097.663991	323023.640654	710.950242

Table 5.10: Results for Backslash Operator (2cubes_sphere).

	i7-10510U	**GTX 1650 Max-Q**	**i7-4710HQ**	**GTX 970M**
Time taken for A\b (s)	0.692094	536.209117	1.770188	623.184345
Gigaflops	1007044.773256	1299.809203	393726.284777	1118.400279

5.3 Data Handling Capability of GPUs

The PCI bus governs the data transfer [34], and both the GeForce GTX 1650 with Max-Q Design and GeForce GTX 970M support PCIe 3.0 with 16 lanes [39, 40]. In principle, a GPU adhering to PCIe 3.0 specifications provides a theoretical bandwidth of 1 GB/s per lane in each direction [47]. Consequently, both GPUs collectively offer a theoretical maximum bandwidth of 16 GB/s per direction. This segment of evaluation has provided three sets of results that are visually presented in Figures 5.13 to 5.18, with each platform's peak performance distinctly marked. Furthermore, all the peak performance data is summarized in Table 5.11 for reference.

5.3.1 Data Transmission and Retrieval Bandwidth

Figures 5.13 and 5.14 depict the data transfer bandwidth between the two CPU-GPU configurations, highlighting the maximum transfer speeds on each platform with a circle. In Figure 5.13, featuring the Core i7-10510U and GeForce GTX 1650 with Max-Q Design, the data transmission speed from the CPU to the GPU consistently outpaces the speed of data retrieval from the GPU to the CPU across various data sizes. However, for notably small data sizes, both transmission and retrieval speeds remain below 1 GB/s. Yet, once the data size exceeds approximately 4 megabytes (MB), both transmission and retrieval speeds notably escalate to around 2.4 GB/s.

Examining Figure 5.14, showcasing the Core i7-4710HQ and GeForce GTX 970M, reveals a similar trend in data transfer speeds as in the other CPU-GPU configuration, which includes the Core i7-10510U and GeForce GTX 1650 with Max-Q Design. The speed of data transmission from the CPU to the GPU generally surpasses the speed of data retrieval from the GPU to the CPU across all data sizes. For small data sizes, the data transmission speed from the CPU to the GPU hovers around 1 GB/s and the data retrieval speed from the GPU to the CPU remains below 1 GB/s. However, once the data size exceeds approximately 2 MB, both transmission and retrieval speeds undergo a significant increase, reaching 3.5 GB/s and 2.8 GB/s, respectively.

Both Figures 5.13 and 5.14 portray a similar pattern, where overheads take precedence when dealing with small data set sizes, and as the data size increases, the PCI bus becomes the limiting factor causing the transfer speed to hover around a certain value [34].

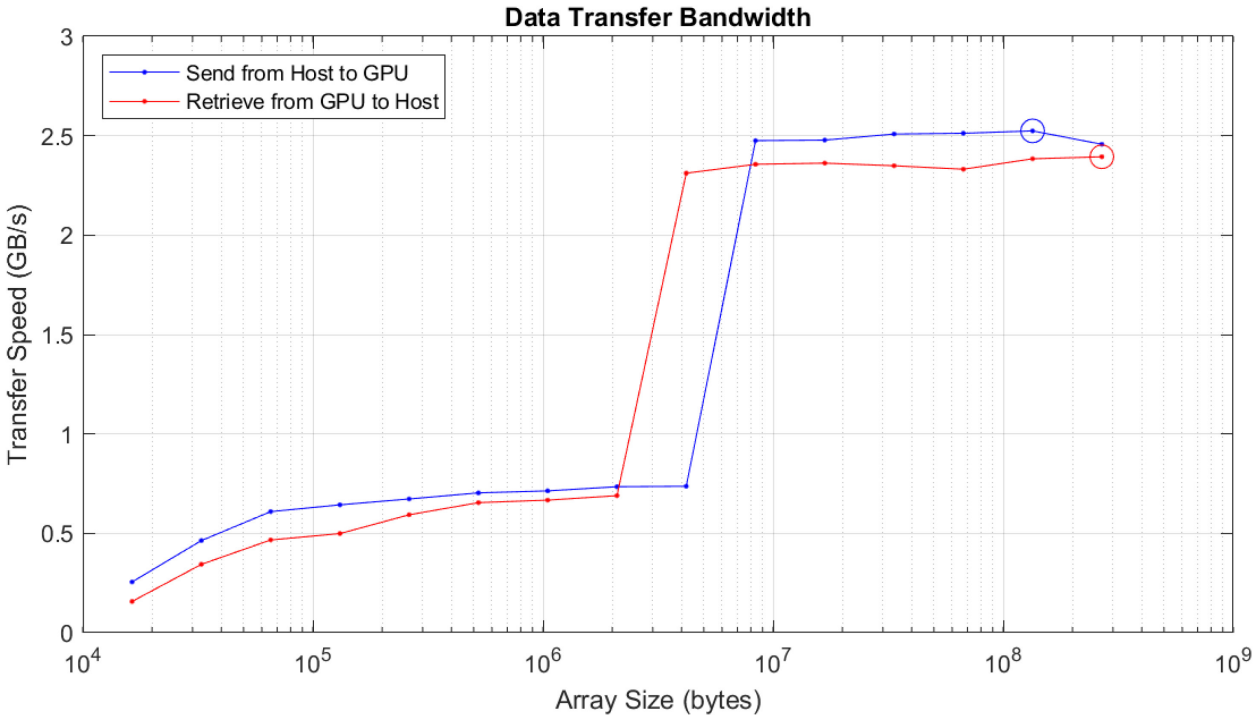

Figure 5.13: Data Transfer Bandwidth between i7-10510U and GTX 1650 with Max-Q.

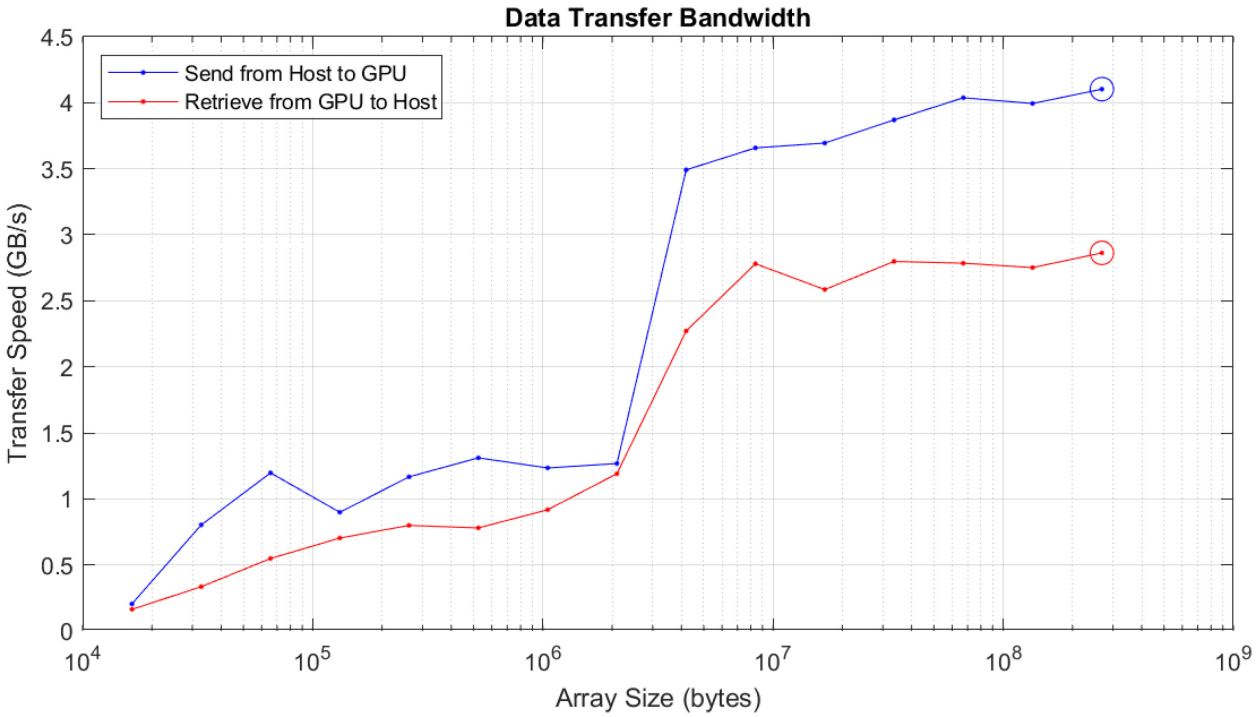

Figure 5.14: Data Transfer Bandwidth between i7-4710HQ and GTX 970M.

5.3.2 Read and Write Data Bandwidth

The outcomes of the `plus` function, which involves one read and one write for each floating-point operation, are presented in Figures 5.15 and 5.16. Figure 5.15 showcases the results of operations conducted on the Core i7-10510U and GeForce GTX 1650 with Max-Q Design, while Figure 5.16 displays the results of operations on the Core i7-4710HQ and GeForce GTX 970M. The maximum speed on each platform is marked with a circle.

Upon examining Figure 5.15, the read-write speed on the Core i7-10510U exhibits a slightly faster average speed than the read-write speed on the GeForce GTX 1650 with Max-Q Design for small data sizes, reaching a peak speed of 104.7 GB/s. However, as the data size increases, the read-write speed on the Core i7-10510U falls below that of the GeForce GTX 1650 with Max-Q Design, maintaining around 12 GB/s for a range of large data sizes. In contrast, the read-write speed on the GeForce GTX 1650 with Max-Q Design continues to steadily increase, peaking at 94.09 GB/s.

In Figure 5.16, a similar pattern to Figure 5.15 is observed. The read-write speed on the Core i7-4710HQ also demonstrates a faster average speed than that on the GeForce GTX 970M for small data sizes, achieving a peak speed of 61.22 GB/s. As the data size increases, the read-write speed on the Core i7-4710HQ experiences a decline, becoming slower than the read-write speed on the GeForce GTX 970M and maintaining around 7 GB/s for a range of large data sizes. Unlike the steady increase seen on the GeForce GTX 1650 with Max-Q Design, the read-write speed on the GeForce GTX 970M exhibits a slot initial increase followed by a rapid leap from 11.6 GB/s to 93.6 GB/s at a data size of approximately 33 MB. It continues to rise with the data size, reaching a peak of 101.45 GB/s.

Comparing Figures 5.15 and 5.16 to Figures 5.13 and 5.14 reveals that the mobile GPUs generally exhibit faster read and write speeds to their memory compared to retrieving data from the host. Hence, minimizing the number of memory transfers between the host and GPU can save time and enhance efficiency. Additionally, transferring data to the GPU for computation, allowing the GPU to perform as much computation as possible before returning the data to the host, proves to be advantageous [34].

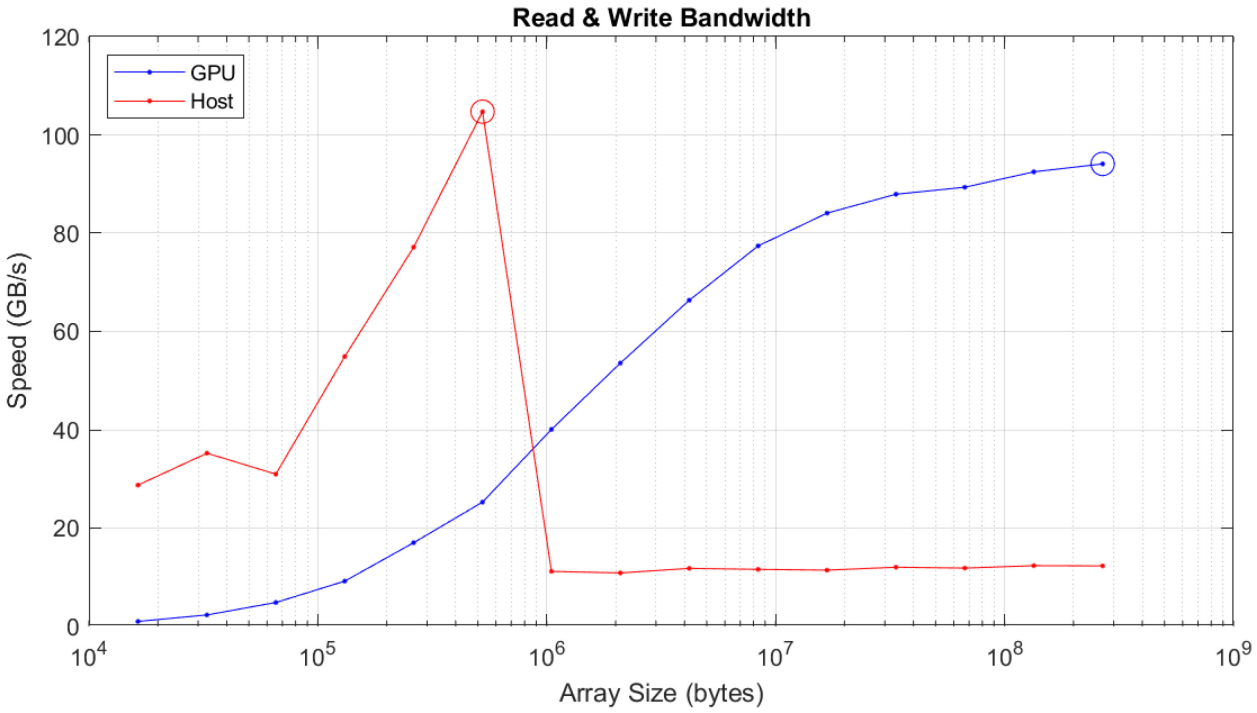

Figure 5.15: Read-Write Bandwidth on i7-10510U and GTX 1650 with Max-Q.

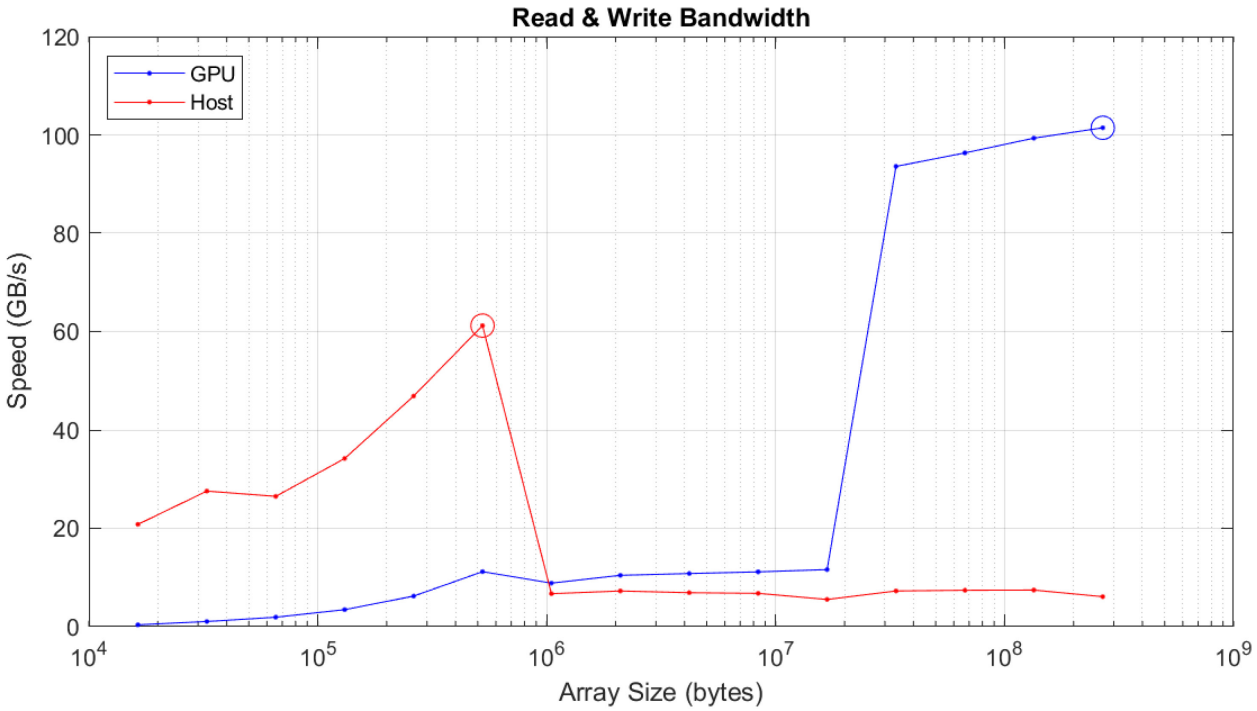

Figure 5.16: Read-Write Bandwidth on i7-4710HQ and GTX 970M.

5.3.3 Calculation Rate of Intensive Operations

Figure 5.17 illustrates the result of the double-precision matrix-matrix multiplication operation on the Core i7-10510U and GeForce GTX 1650 with Max-Q Design, while Figure 5.18 displays the result of the same operation on the Core i7-4710HQ and GeForce GTX 970M. The highest calculation rate in gigaflops is circled on each platform.

Both Figures 5.17 and 5.18 reveal a consistent pattern where the CPUs outperform the GPUs for all matrix sizes. When the matrix size is relatively small, both the CPUs and GPUs exhibit lower calculation rate. As the matrix size increases, the calculation rate on both the CPUs and GPUs shows an upward trend. In Figure 5.17, the Core i7-10510U achieves a peak calculation rate of 155.78 GFLOPS, while the GeForce GTX 1650 with Max-Q Design attains a peak calculation rate of 113.29 GFLOPS. In Figure 5.18, the Core i7-4710HQ records a peak calculation rate of 130.2 GFLOPS, whereas the GeForce GTX 970M achieves a peak calculation rate of 76.06 GFLOPS.

The findings indicate that the mobile GPUs excel in performing calculations more rapidly when dealing with sufficiently large data sizes compared to smaller ones. Nevertheless, the overall performance lags behind that of the CPUs due to the restricted memory bandwidth, power and thermal constraints of mobile GPUs [43], aligning with a consistent trend observed in previous sub-chapters (Chapters 5.1.1 and 5.2.1). Despite the performance limitations of mobile GPUs, it remains noticeable that the GPU achieves higher GFLOPS when operating at higher level of saturation as the overhead linked to initiating and managing parallel tasks on the GPU decreases [13].

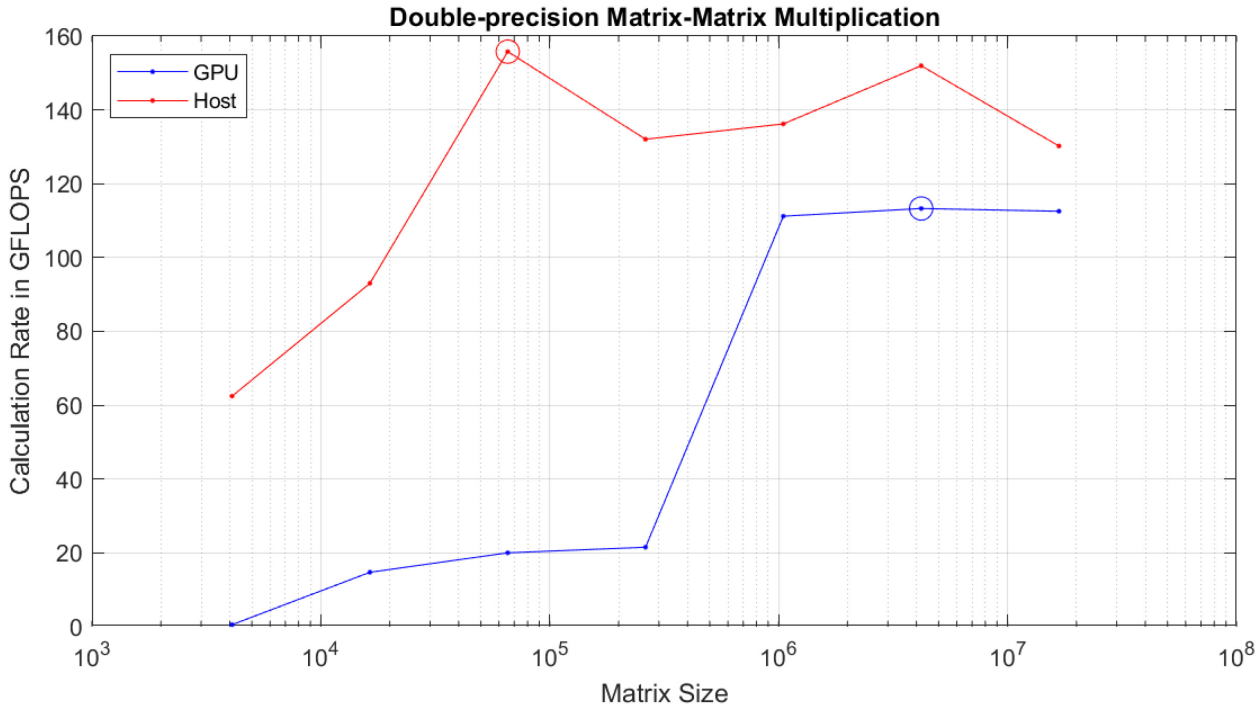

Figure 5.17: Rate of Matrix Multiplication Operation on i7-10510U and GTX 1650 with Max-Q.

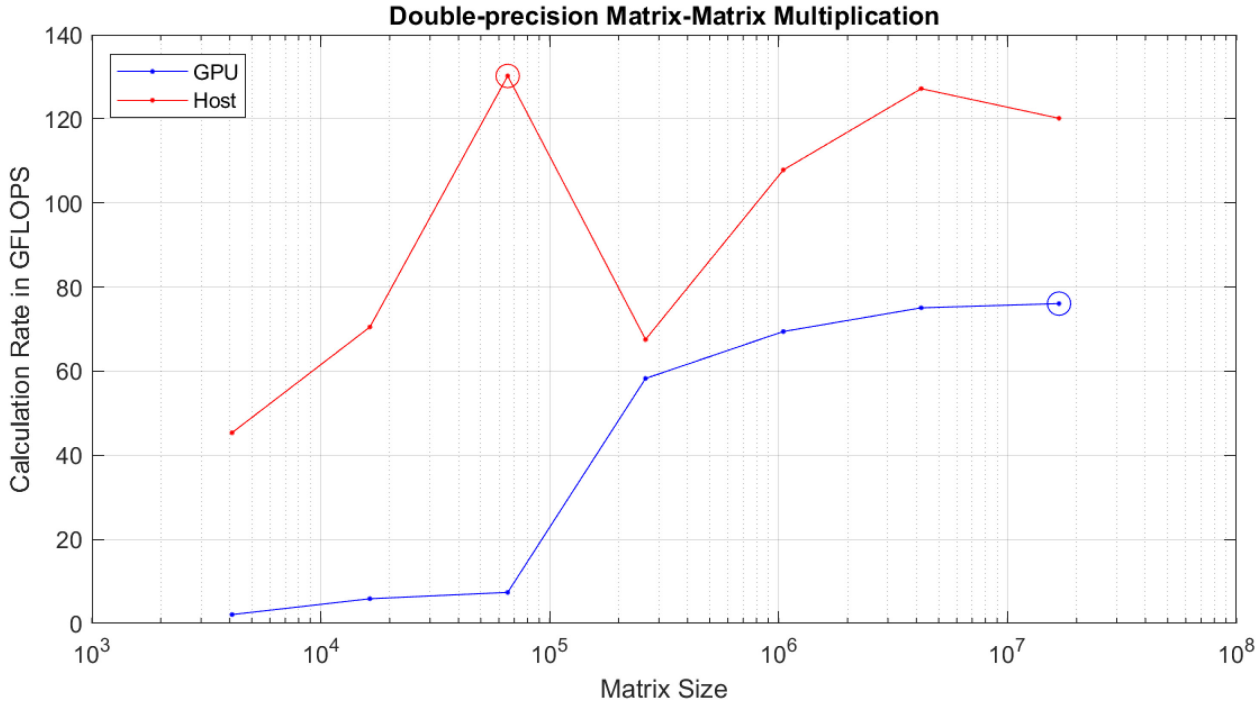

Figure 5.18: Rate of Matrix Multiplication Operation on i7-4710HQ and GTX 970M.

Table 5.11: Results of Data Handling between CPU and GPUs.

	i7-10510U	**GTX 1650 Max-Q**	**i7-4710HQ**	**GTX 970M**
Peak Send Speed from Host to GPU (GB/s)	2.52417		4.10281	
Peak Gather Speed from GPU to Host (GB/s)	2.39407		2.86279	
Peak Read-Write Speed (GB/s)	104.722	94.0939	61.2189	101.449
Peak Operation Rate (GFLOPS)	155.8	113.3	130.2	76.1

5.4 Resource Contention on CPUs

The assessment of resource contention on CPUs has provided two sets of results for each CPU and GPU configuration—one for varying the number of processes and another for varying the data size [37]. Moreover, the outcomes from this evaluation help to understand the behaviour of the results obtained in Chapter 5.1.2.

5.4.1 Varying Number of Processes

This phase of the evaluation encompasses three distinct operations: summation, DFFT, and matrix-matrix multiplication. Figure 5.19 is derived from the test conducted on the Core i7-10510U, while Figure 5.20 corresponds to the Core i7-4710HQ. Given that both CPUs boast four cores [44, 45], the range of the parallel workers spans from one to four. The speedup, depicted in Figures 5.19 and 5.20, is calculated using a consistent formula that involves determining the ratio of the time taken with the minimum number of parallel workers (one) to the time taken with the specified number of parallel workers (ranging from one to four). This result is then multiplied by the number of parallel workers employed, which also ranges from one to four, providing a scaled

representation on the graph. Additionally, the numerical results necessary for plotting both Figures 5.19 and 5.20 are exhaustively detailed in Tables 5.12 to 5.14.

Upon scrutiny of Figures 5.19 and 5.20, it becomes obvious that summation operations, being computationally lightweight, exhibit pronounced resource contention, as reflected in the gradual increase in speedup with an increase in the number of processes, a trend consistent for both CPUs. In Figure 5.19, the Core i7-10510U achieves a speedup of 1.59 with four parallel workers, while in Figure 5.20, the Core i7-4710HQ attains a speedup of 1.39 with the same number of parallel workers. Consequently, executing multiple lightweight operations concurrently requires more time than a single execution of such an operation on a CPU [37].

On the contrary, DFFT operations, being more computationally intensive than summation operations, showcase enhanced speedup performance on both CPUs. Figure 5.19 indicates that the Core i7-10510U achieves a speedup of 1.97 with four parallel workers, while Figure 5.20 shows that the Core i7-4710HQ attains a speedup of 2.27 under identical conditions. This improved speedup performance suggests a reduction in resource contention. Thus, DFFT operations do not display the same performance degradation as summation operations when multiple calls are concurrently executed [37].

Lastly, matrix-matrix multiplication operations demonstrate the highest speedup performance as the number of processes increases. Figure 5.19 reveals that the Core i7-10510U attains a speedup of 3 with four parallel workers, and Figure 5.20 indicates that the Core i7-4710HQ achieves a speedup of 2.53 under the same conditions. This efficiency is attributed to the regular memory access in matrix-matrix multiplication, making it highly effective for parallel execution on a multicore platform [37].

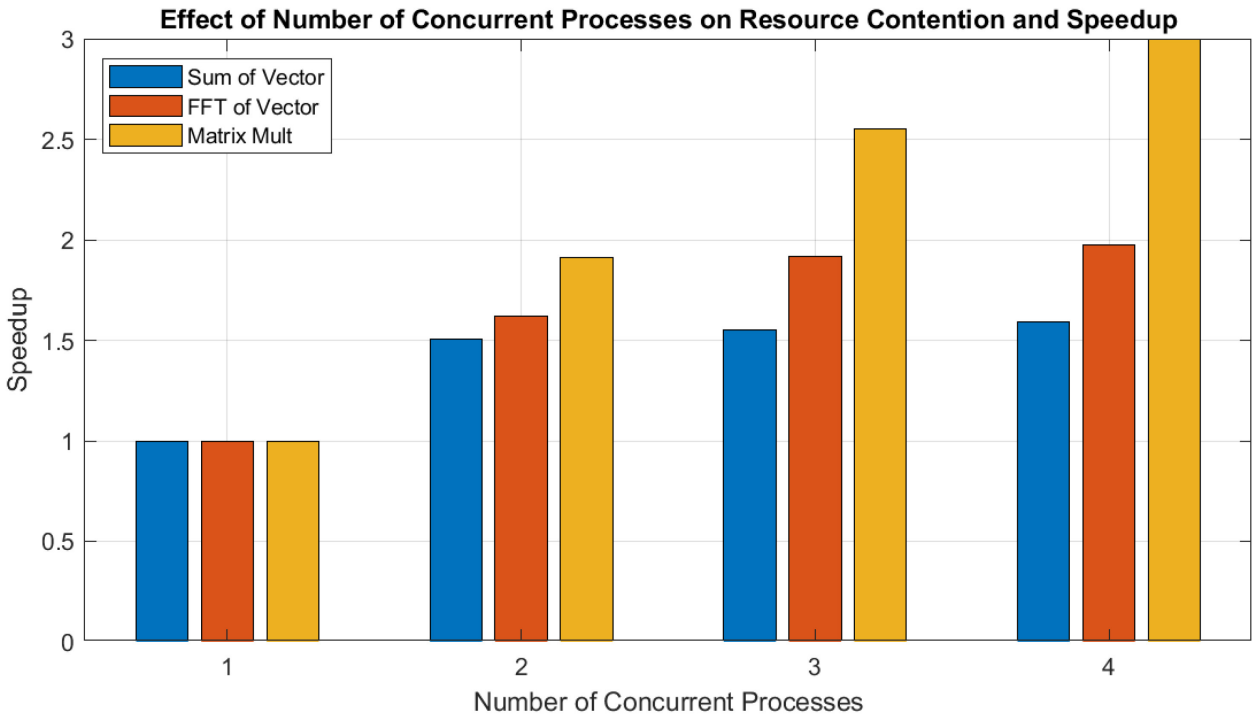

Figure 5.19: Effect of Concurrent Processes on Resource Contention on Core i7-10510U.

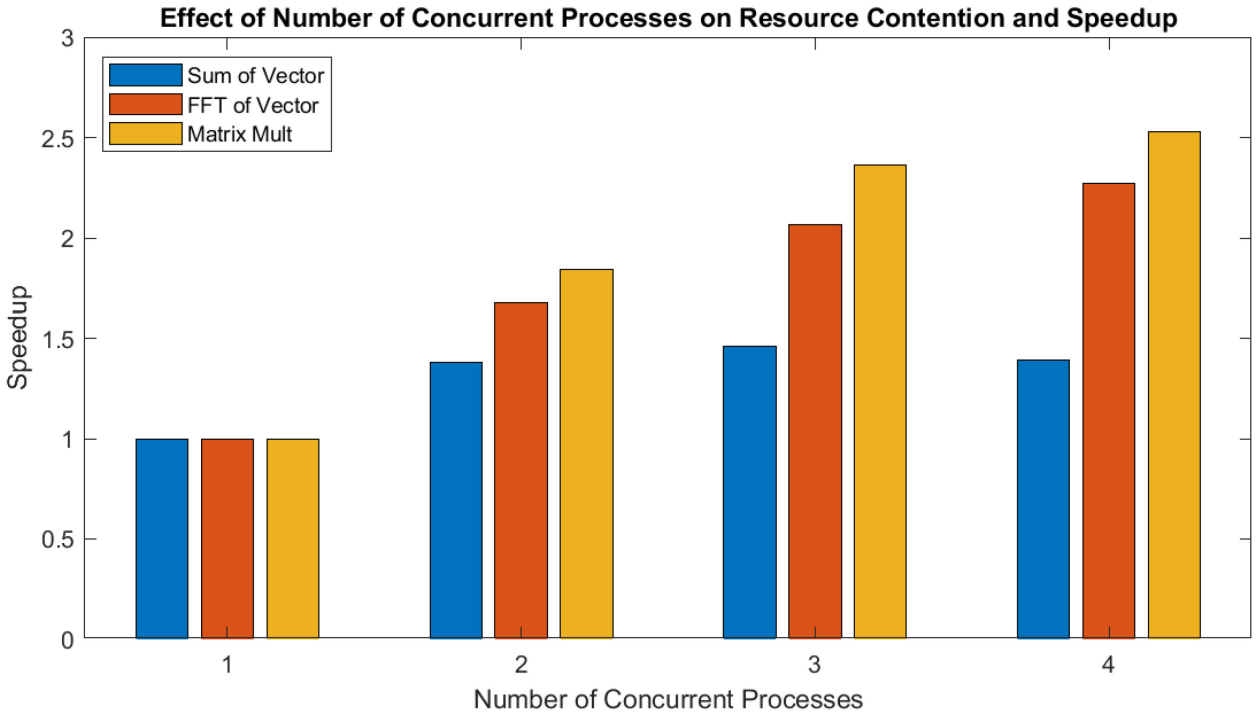

Figure 5.20: Effect of Concurrent Processes on Resource Contention on Core i7-4710HQ.

Table 5.12: Results for Summation Operations on an array of 2048^2.

Number of Parallel Workers	**Core i7-10510U Time taken (s)**	**Core i7-4710HQ Time taken (s)**
1	0.208519	0.253275
2	0.277223	0.366846
3	0.403255	0.521152
4	0.523741	0.728668

Table 5.13: Results for DFFT Operations on a vector of 2048^2.

Number of Parallel Workers	**Core i7-10510U Time taken (s)**	**Core i7-4710HQ Time taken (s)**
1	0.708381	1.025814
2	0.873899	1.223243
3	1.108064	1.491441
4	1.435778	1.805978

Table 5.14: Results for Matrix Multiplication Operations of 2048×2048.

Number of Parallel Workers	**Core i7-10510U Time taken (s)**	**Core i7-4710HQ Time taken (s)**
1	0.327313	0.401454
2	0.342751	0.436009
3	0.384471	0.509765
4	0.436631	0.634544

5.4.2 Varying Data Size

When evaluating resource contention with varying data sizes, additional operations namely LU decomposition, SVD, and eigenvalue computation, are considered in conjunction with the initial three operations—summation, DFFT, and matrix-matrix multiplication. Figures 5.21 and 5.22 present the results obtained from the Core i7-10510U and Core i7-4710HQ, respectively. The speedup, illustrated in both figures, is computed using the same formula that involves getting the ratio of the time taken with the minimum number of parallel workers (one) to the time taken with the maximum number of parallel workers (four). The result is then multiplied by the number of parallel workers employed, which is four in this instance. Both CPUs ideally exhibit a speedup of 4, corresponding to the number of cores each CPU possesses [37]. Furthermore, the numerical results essential for plotting both Figures 5.21 and 5.22 are exhaustively detailed in Tables 5.15 and 5.16.

In Figure 5.21, showcasing the Core i7-10510U, summation and SVD operations exhibit a declining trend as the number of elements per parallel worker increases. On the contrary, matrix-matrix multiplication and LU decomposition operations demonstrate an ascending trend with an increase in the number of elements per parallel worker. The DFFT operation maintains a consistent speedup across all number of elements per process. Lastly, the eigenvalue operation displays an inconsistent trend, initially showing an upward trajectory followed by a subsequent downward trend.

Similarly, in Figure 5.22, featuring the Core i7-4710HQ, the summation and SVD operations depict a declining trend with an increase in the number of elements per parallel worker. In contrast, the matrix-matrix multiplication operation displays an upward trend under the same conditions. Both the DFFT and LU decomposition operations maintain a constant speedup across all number of elements per process. The eigenvalue operation, akin to Figure 5.21, exhibits an inconsistent trend, initially ascending and later descending. In summary, both CPUs exhibit similar behaviour across all operations, except for the LU decomposition operation, which displays divergent patterns.

Upon examining the outcomes, it becomes evident that for small data sizes, the functions operate efficiently within the CPU cache, yielding a relatively commendable speedup. Contrarily, as the data size surpasses the capacity of the CPU cache, a decline in performance attributable to contention for memory access becomes apparent [37]. This trend of performance degradation due to an increase in data size is also observable in Chapter 5.1.2 where the loaded problems, thermal1 and 2cubes_sphere [41, 42], are considerably larger than the generated matrices.

Figure 5.21: Effect of Data Size on Resource Contention on Core i7-10510U.

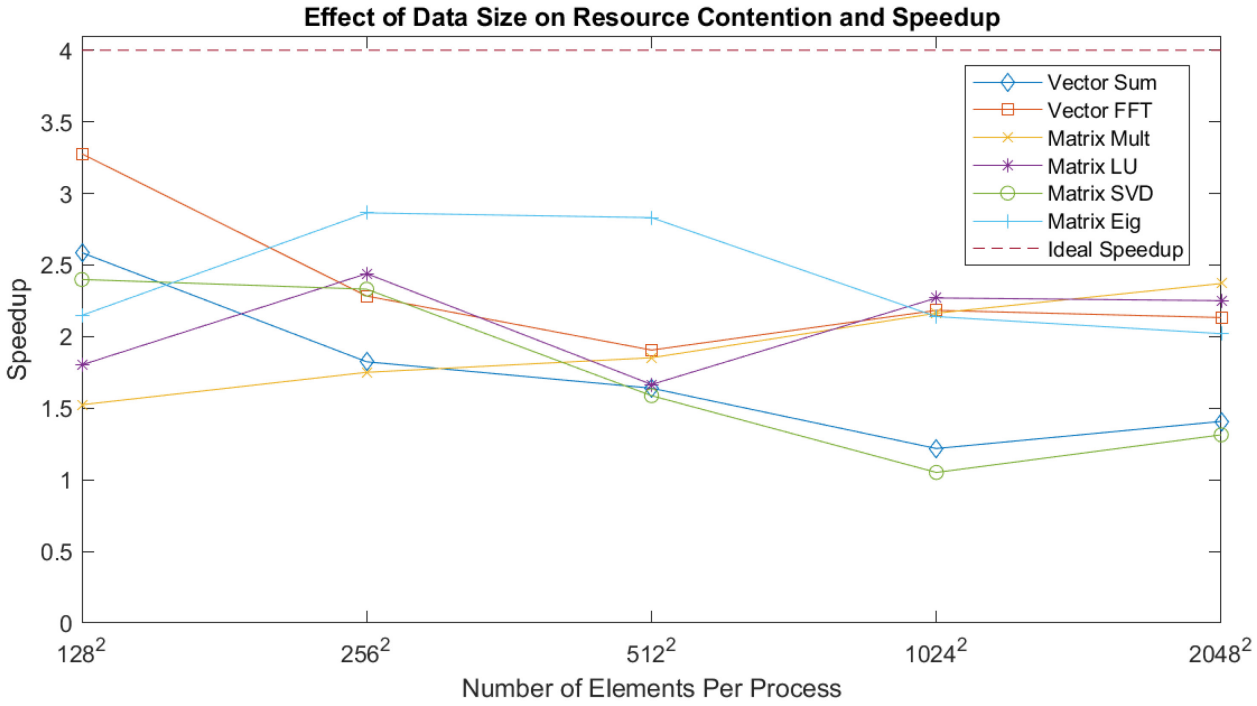

Figure 5.22: Effect of Data Size on Resource Contention on Core i7-4710HQ.

Table 5.15: Time Taken (s) for Various Operations on Varying Data Size with 1 Parallel Worker.

	Data Size / **Operation**	128×128	256×256	512×512	1024×1024	2048×2048
Core i7-10510U	Sum	0.000417	0.001388	0.005975	0.037754	0.200022
	DFFT	0.001558	0.008644	0.037326	0.198239	0.712986
	Matrix Mult	0.000089	0.000793	0.005225	0.041985	0.313504
	LU	0.000206	0.000420	0.002965	0.019379	0.137334
	SVD	0.000735	0.003922	0.023647	0.185570	2.500387
	Eig	0.004672	0.021461	0.151504	0.584634	3.304784
Core i7-4710HQ	Sum	0.000509	0.001979	0.008301	0.054400	0.248218
	DFFT	0.003515	0.013114	0.053108	0.274265	0.981628
	Matrix Mult	0.000139	0.000913	0.006842	0.053766	0.404726
	LU	0.000163	0.001130	0.005080	0.029384	0.191172
	SVD	0.001262	0.007012	0.040569	0.391743	4.187507
	Eig	0.007526	0.035603	0.246028	1.069602	5.409669

Table 5.16: Time Taken (s) for Various Operations on Varying Data Size with 4 Parallel Workers.

	Data Size / **Operation**	128×128	256×256	512×512	1024×1024	2048×2048
Core i7-10510U	Sum	0.000691	0.002788	0.011428	0.116797	0.521592
	DFFT	0.002003	0.015085	0.071033	0.364284	1.459302
	Matrix Mult	0.000224	0.001717	0.007689	0.055448	0.439225
	LU	0.000127	0.000986	0.006442	0.039014	0.217332
	SVD	0.001173	0.006619	0.038824	0.833646	9.560515
	Eig	0.008513	0.025307	0.188193	1.117153	7.939985
Core i7-4710HQ	Sum	0.000787	0.004339	0.020244	0.178574	0.705176
	DFFT	0.004290	0.022973	0.111446	0.502426	1.840714
	Matrix Mult	0.000364	0.002086	0.014772	0.099395	0.682797
	LU	0.000361	0.001853	0.012198	0.051780	0.339684
	SVD	0.002104	0.012032	0.102208	1.490041	12.745665
	Eig	0.014010	0.049717	0.347629	1.999281	10.708117

5.5 MATLAB's GPUBench

The GPUBench tool automatically generates a report following the execution of matrix-matrix multiplication, backslash DFFT operations in both single- and double-precision modes [38]. This report includes a performance comparison of the tested CPUs and GPUs against the performance of other GPUs. Figure 5.23 summarizes the performance of the first CPU-GPU configuration, featuring the Core i7-10510U and GeForce GTX 1650 with Max-Q Design, while Figure 5.24 outlines the performance summary of the second CPU-GPU configuration, comprising the Core i7-4710HQ and GeForce GTX 970M. Furthermore, Table 5.17 shows the detailed numerical results for both CPU-GPU configurations, with the tested CPUs and GPUs highlighted in bold for easy reference.

Examining both Figures 5.23 and 5.24 reveals that GPUs exhibit significantly higher GFLOPS when handling single-precision variables. However, in the case of double-precision variables, the GPUs either lag behind in GFLOPS or show comparable performance to the corresponding CPUs. These findings align with the observation made in the assessment conducted in Chapter 5.2.1, specifically regarding the backslash operation. Moreover, in the earlier sub-chapters (Chapters 5.1.1, 5.2.1, and 5.3.3), focusing on double-precision data types, also highlight that the mobile GPUs do not excel due to similar factors such as limited bandwidth, power constraints, and thermal limitations [43].

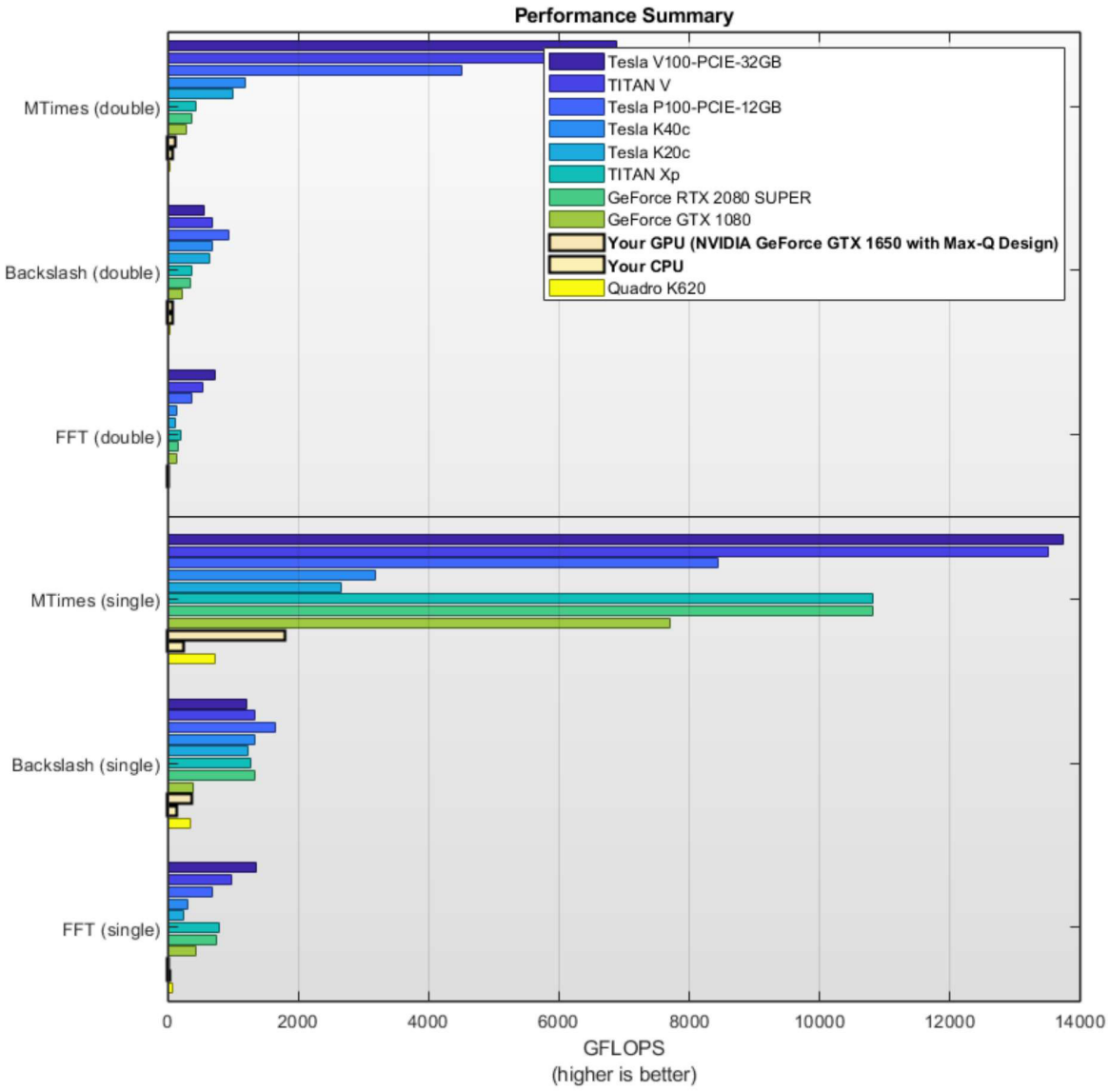

Figure 5.23: Performance Summary of i7-10510U and GTX 1650 with Max-Q.

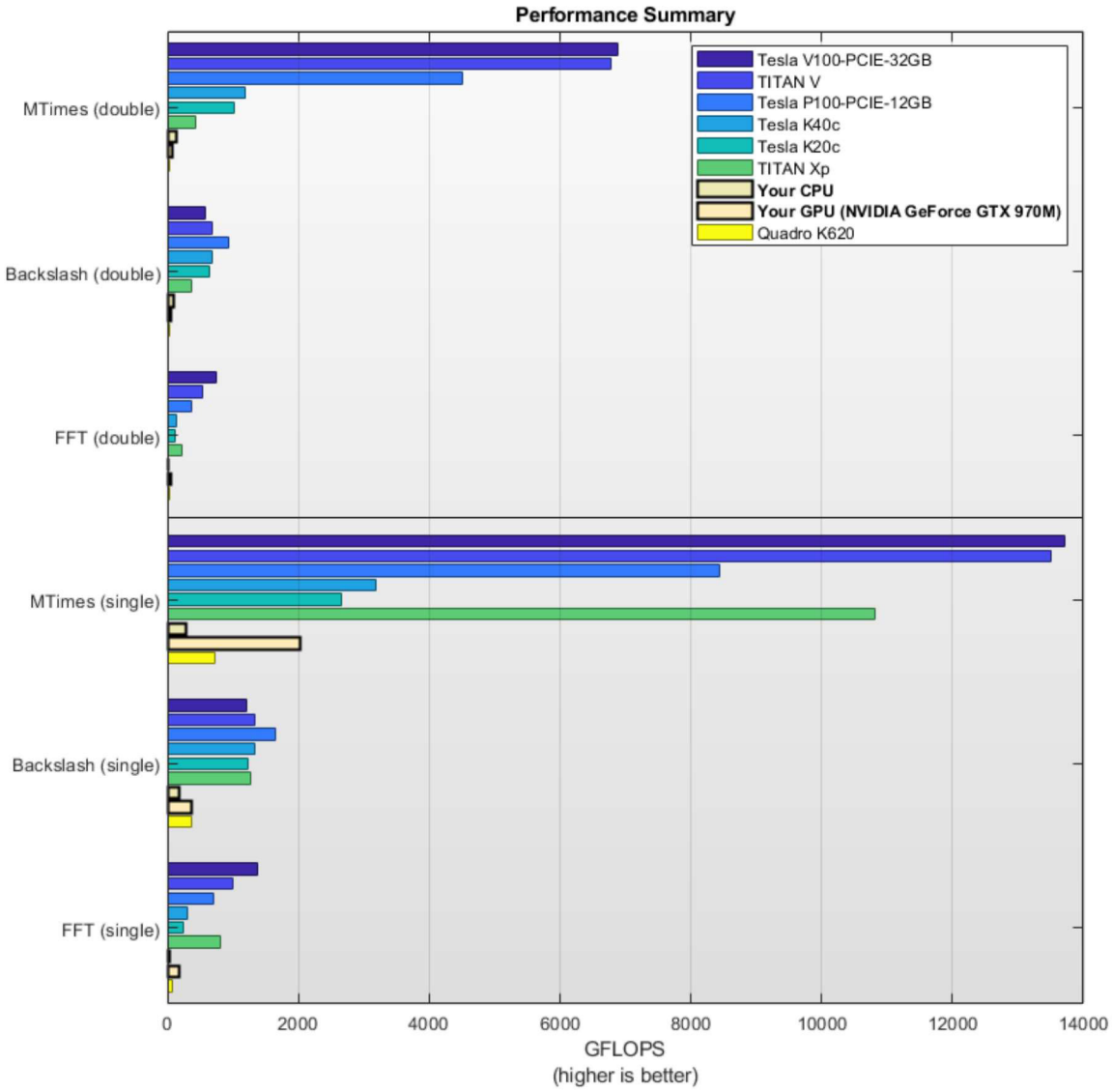

Figure 5.24: Performance Summary of i7-4710HQ and GTX 970M.

Table 5.17: Summary of All Tested CPUs and GPUs.

	Double-precision Results (GFLOPS)			**Single-precision Results (GFLOPS)**		
	Matrix Multi	**Backslash**	**DFFT**	**Matrix Multi**	**Backslash**	**DFFT**
Tesla V100-PCIE-32GB	6884.95	563.73	728.71	13727.99	1210.42	1365.11
TITAN V	6779.73	674.40	534.65	13515.42	1336.39	985.36
Tesla P100-PCIE-12GB	4510.03	929.00	357.65	8435.34	1647.83	687.13
Tesla K40c	1189.54	677.12	135.88	3187.76	1334.17	294.86
Tesla K20c	1004.06	641.42	106.09	2657.01	1230.28	235.20
TITAN Xp	421.00	369.32	209.45	10823.05	1272.06	797.17
GeForce RTX 2080 Super	373.37	345.32	164.30	10813.12	1330.64	746.20
GeForce GTX 1080	280.84	223.05	137.66	7707.01	399.37	424.60
GeForce GTX 1650 with Max-Q Design	111.41	83.54	3.85	1807.22	371.49	18.53
Core i7-10510U	79.20	67.66	9.95	246.96	135.04	32.53
GeForce GTX 970M	74.97	50.81	38.93	2014.80	369.06	179.78
Core i7-4710HQ	126.50	86.24	9.92	265.75	180.26	29.36
Quadra K620	25.45	22.77	12.75	716.71	350.31	75.00

Chapter 6
Conclusion and Future Work

In this thesis, we conducted a comprehensive analysis of GPU parallelization performance, specifically focusing on NVIDIA mobile graphics chips, utilizing MATLAB with PCT instead of APIs like CUDA. Our investigation involved the implementation of the ICPCG method and the backslash operation within mobile devices, such as laptops. Furthermore, we examined the data handling capabilities of the mobile GPUs, assessed resource contention, and utilized the GPUBench tool developed by the MathWorks PCT Team.

Our finding suggests that mobile NVIDIA GPUs, particularly those with Turing and Maxwell 2.0 architectures, do not offer substantial advantages in enhancing the efficiency of the ICPCG method when employing MATLAB PCT, especially in scenarios involving double-precision variables. The absence of CUDA may contribute to suboptimal GPU resource utilization since MATLAB PCT lacks options for developers to explicitly assign tasks to GPUs [5], in contrast to CUDA. As a result, developers rely on the toolbox for task allocation, limiting the optimization of code for parallel execution.

Moreover, it is noteworthy that MATLAB currently does not support sparse single-precision data types [48]. This limitation forces GPUs to handle large sparse data in double-precision only, contributing to a slower computation rate on mobile GPUs. While the tested mobile NVIDIA GPUs, including the GeForce GTX 1650 with Max-Q Design and GeForce GTX 970M, did not outperform the tested Intel® CPUs, featuring Core™ i7-10510U and Core™ i7-4710HQ, our results shed light on the constraints related to implementing iterative solvers in the MATLAB environment on mobile GPUs.

Understanding the underlying reasons behind these results, we identify potential directions for future work. Firstly, enhancing MATLAB to support large sparse single-precision matrices could significantly improve the efficiency of GPU operations, potentially surpassing CPU performance. Additionally, exploring advancements in MATLAB toolboxes to enable developers to explicitly parallelize their code for optimal efficiency is crucial. Furthermore, considering the

continuous evolution of GPUs and PCI buses, improvements in memory bandwidth, power constraints, and thermal limitations are anticipated. These advancements may allow for increased data storage and processing within GPU memory, alleviating performance bottlenecks. Future research in these directions holds the promise of overcoming current limitations and unlocking the full potential of mobile GPU parallelization in MATLAB.

Appendix

Table A.1: Specifications of Tested GPUs.

Graphics Processor	**NVIDIA GeForce GTX 1650 with Max-Q Design[40]**	**NVIDIA GeForce GTX 970M[39]**
Architecture	Turing	Maxwell 2.0
GPU Name	TU117	GM204
Process Size	12 nm	28nm
Transistors	4,700 million	5,200 million
Density	23.5 M / mm^2	13.1 M / mm^2
Die Size	200 mm^2	398 mm^2
Bus Interface	PCIe 3.0 x16	MXM-B (3.0 x16)
Release Date	23 April 2019	7 October 2014
Memory		
Memory Size	4 GB	3 GB
Memory Type	GDDR5	GDDR5
Memory Bus Width	128-bit	192-bit
Memory Bandwidth	112.1 GB/s	120.3 GB/s
Render Config		
Cores	1024	1280
TMUs	64	80
ROPs	32	48
SM Count	16	10
L1 Cache	64 KB (per SM)	48 KB (per SM)
L2 Cache	1024 KB	1536 KB
Clock Speeds		
Base Clock	1020 MHz	924 MHz
Boost Clock	1245 MHz	1038 MHz

Memory Clock	1751 MHz, 7 Gbps effective	1253 MHz, 5 Gbps effective
Graphics Features		
DirectX	12_1	12_1
OpenGL	4.6	4.6
OpenCL	3.0	3.0
Vulkan	1.3	1.3
CUDA	7.5	5.2
Shader Model	6.7	6.7
Power Consumption	35 Watt	81 Watt

Table A.2: Specifications of Tested CPUs.

Central Processor	**Intel® Core™ i7-10510U[45]**	**Intel® Core™ i7-4710HQ[44]**
Essentials		
Product Collection	10th Generation Intel® Core™ i7 Processors	4th Generation Intel® Core™ i7 Processors
Code Name	Products formerly Comet Lake	Products formerly Haswell
Vertical Segment	Mobile	Mobile
Processor Number	i7-10510U	i7-4710HQ
Lithography	14 nm	22 nm
Launch Date	Q3’19	Q2’14
CPU Specifications		
Total Cores	4	4
Total Threads	8	8
Max Turbo Frequency	4.90 GHz	3.50 GHz
Processor Base Frequency	1.80 GHz	2.50 GHz
Cache	8 MB Intel® Smart Cache	6 MB Intel® Smart Cache
Bus Speed	4 GT/s	5 GT/s

TDP	15 W	47 W
Memory Specifications		
Max Memory Size (dependent on memory type)	64 GB	32 GB
Memory Types	DDR4-2666, LPDDR3-2133, LPDDR4-2933	DDR3L 1333/1600
Max # of Memory Channels	2	2
Max Memory Bandwidth	45.8 GB/s	25.6 GB/s
ECC Memory	No	No
GPU Specifications		
Processor Graphics	Intel® UHD Graphics for 10th Gen Intel® Processors	Intel® HD Graphics 4600
Graphics Base Frequency	300 MHz	400 MHz
Graphics Max Dynamic Frequency	1.15 GHz	1.20 GHz
Graphics Video Max Memory	32 GB	2 GB
Device ID	0x9B41/0x9BCC	0x416
Expansion Options		
PCI Express Revision	3.0	3.0
Max # of PCI Express Lanes	16	16

Bibliography

[1] J. Reese and S. Zaranek. "GPU Programming in MATLAB." https://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html (accessed 2023-12-03).

[2] J. Peddie, *The History of the GPU-New Developments*, First ed. Springer Cham, 2023, p. 410.

[3] S. K. David, "The incomplete Cholesky—conjugate gradient method for the iterative solution of systems of linear equations," *Journal of Computational Physics,* vol. 26, no. 1, pp. 43-65, 1978, doi: https://doi.org/10.1016/0021-9991(78)90098-0.

[4] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations* (Frontiers in Applied Mathematics). Society for Industrial and Applied Mathematics, 1995, p. 169.

[5] MathWorks, "Parallel Computing Toolbox User's Guide."

[6] NVIDIA. "CUDA C++ Programming Guide." NVIDIA. https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html (accessed 2023-11-27).

[7] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards,* vol. 49, pp. 409-435, 1952.

[8] R. M. Freund, "The Steepest Descent Algorithm for Unconstrained Optimization and a Bisection Line-search Method," *Journal of Massachusetts Institute of Technology. United States of America,* vol. 131, 2004.

[9] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain," Carnegie Mellon University, 1994. Accessed: 2023-12-13.

[10] R. M. Freund, "The Steepest Descent Algorithm for Unconstrained Optimization," *Journal of Massachusetts Institute of Technology. United States of America,* 2014.

[11] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second ed. Society for Industrial and Applied Mathematics, 2003, p. 537.

[12] W. Ford, *Numerical Linear Algebra with Applications using MATLAB*. 2014.

[13] M. J. Mišić, Đ. M. Đurđević, and M. V. Tomašević, "Evolution and trends in GPU computing," in *2012 Proceedings of the 35th International Convention MIPRO*, Opatija, Croatia, 2012: IEEE, pp. 289-294.

[14] NVIDIA. "GPU Performance Background." NVIDIA. https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-arch (accessed 2023-11-27).

[15] MathWorks. "Perform parallel computations on multicore computers, GPUs, and computer clusters." https://www.mathworks.com/help/parallel-computing/ (accessed 2023-12-05).

[16] MathWorks. "Query or select a GPU device." https://www.mathworks.com/help/parallel-computing/parallel.gpu.gpudevice.html (accessed 2023-12-06).

[17] MathWorks. "Incomplete Cholesky factorization." https://www.mathworks.com/help/matlab/ref/ichol.html?s_tid=doc_ta (accessed 2023-12-05).

[18] MathWorks. "Solve system of linear equations — preconditioned conjugate gradient method." https://www.mathworks.com/help/matlab/ref/pcg.html?s_tid=doc_ta (accessed 2023-12-05).

[19] MathWorks. "Start stopwatch timer." https://www.mathworks.com/help/matlab/ref/tic.html?s_tid=doc_ta (accessed 2023-12-06).

[20] MathWorks. "Read elapsed time from stopwatch." https://www.mathworks.com/help/matlab/ref/toc.html (accessed 2023-12-06).

[21] MathWorks. "Array stored on GPU." https://www.mathworks.com/help/parallel-computing/gpuarray.html (accessed 2023-12-06).

[22] MathWorks. "Time required to run function on GPU." https://www.mathworks.com/help/parallel-computing/gputimeit.html (accessed 2023-12-06).

[23] MathWorks. "Create Function Handle." https://www.mathworks.com/help/matlab/matlab_prog/creating-a-function-handle.html (accessed 2023-12-06).

[24] MathWorks. "Wait for futures to complete." https://www.mathworks.com/help/matlab/ref/parallel.future.wait.html?s_tid=doc_ta (accessed 2023-12-06).

[25] MathWorks. "Measure and Improve GPU Performance." https://www.mathworks.com/help/parallel-computing/measure-and-improve-gpu-performance.html (accessed 2023-12-04).

[26] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers." University of Tennessee Computer Science Department. https://www.netlib.org/benchmark/hpl/ (accessed 2023-11-27).

[27] MathWorks. "Execute code in parallel on workers of parallel pool." https://www.mathworks.com/help/parallel-computing/spmd.html?s_tid=doc_ta (accessed 2023-12-06).

[28] MathWorks. "Create parallel pool on cluster." https://www.mathworks.com/help/parallel-computing/parpool.html (accessed 2023-12-06).

[29] MathWorks. "Run MATLAB on multicore and multiprocessor machines." https://www.mathworks.com/discovery/matlab-multicore.html (accessed 2023-12-05).

[30] MathWorks. "Solve systems of linear equations Ax=B for x." https://www.mathworks.com/help/matlab/ref/mldivide.html (accessed 2023-12-05).

[31] MathWorks. "Benchmarking A\b on the GPU." https://www.mathworks.com/help/parallel-computing/benchmarking-a-b-on-the-gpu.html (accessed 2023-12-05).

[32] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers," *Concurrency and Computation: Practice and Experience,* vol. 31, no. 6, p. e4460, 2019, doi: https://doi.org/10.1002/cpe.4460.

[33] H. Markus and E. S. Paul, "Block jacobi preconditioning of the conjugate gradient method on a vector processor," *International Journal of Computer Mathematics,* vol. 44, no. 1-4, pp. 71-89, 1992, doi: https://doi.org/10.1080/00207169208804096.

[34] MathWorks. "Measure GPU Performance." https://www.mathworks.com/help/parallel-computing/measuring-gpu-performance.html (accessed 2023-12-04).

[35] MathWorks. "Transfer distributed array, Composite object, or gpuArray object to local workspace." https://www.mathworks.com/help/parallel-computing/gpuarray.gather.html (accessed 2023-12-06).

[36] MathWorks. "Measure time required to run function." https://www.mathworks.com/help/matlab/ref/timeit.html?s_tid=doc_ta (accessed 2023-12-05).

[37] MathWorks. "Resource Contention in Task Parallel Problems." https://www.mathworks.com/help/parallel-computing/resource-contention-in-task-parallel-problems.html (accessed 2023-12-05).

[38] M. P. C. T. Team. "Compare GPUs using standard numerical benchmarks in MATLAB." https://www.mathworks.com/matlabcentral/fileexchange/34080-gpubench?s_tid=srchtitle_support_results_1_gpubench (accessed 2023-12-05).

[39] NVIDIA. "NVIDIA GeForce GTX 970M Specifications." https://www.nvidia.com/en-us/geforce/gaming-laptops/gtx-970m/specifications/ (accessed 2023-12-13).

[40] NVIDIA. "NVIDIA GeForce GTX 1650 Max-Q." https://www.techpowerup.com/gpu-specs/geforce-gtx-1650-max-q.c3383 (accessed 2023-12-13).

[41] D. Schmid and T. Davis. *Schmid/thermal1 unstructured FEM, steady state thermal problem.* [Online]. Available: https://sparse.tamu.edu/Schmid/thermal1

[42] E. Um and T. Davis. *Um/2cubes_sphere FEM, electromagnetics, 2 cubes in a sphere.* [Online]. Available: https://sparse.tamu.edu/Um/2cubes_sphere

[43] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the Limits of GPU Acceleration," presented at the Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism, 2010.

[44] Intel®. "Intel® Core™ i7-4710HQ Processor." https://www.intel.com/content/www/us/en/products/sku/78930/intel-core-i74710hq-processor-6m-cache-up-to-3-50-ghz/specifications.html (accessed 2023-12-13).

[45] Intel®. "Intel® Core™ i7-10510U Processor." https://www.intel.com/content/www/us/en/products/sku/196449/intel-core-i710510u-processor-8m-cache-up-to-4-90-ghz/specifications.html (accessed 2023-12-13).

[46] MathWorks. "Floating-Point Numbers." https://www.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html (accessed 2023-12-06).

[47] PCI-SIG. "PCI Express® 3.0." https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2 (accessed 2023-12-13).

[48] MathWorks. "Create codistributed sparse matrix." https://www.mathworks.com/help/parallel-computing/codistributed.sparse.html?s_tid=doc_ta (accessed 2023-12-06).