Exploring heterogeneity with multicore CPU and integrated GPUs

André Cerqueira
Master in Computer Science
Department of Computer Science
Faculty of Sciences of the University of Porto
2024

Exploring heterogeneity with multicore CPU and integrated GPUs

André Cerqueira

Dissertation carried out as part of the Master in Computer Science

Department of Computer Science

2024

Supervisor
Prof. Dr. Miguel Areias,
Assistant Professor,
Faculty of Science of the University of Porto

Co-supervisor
Prof. Dr. Jorge Barbosa,
Associate Professor,
Faculty of Engineering of the University of Porto

“ I’ve seen things you people wouldn’t believe. Attack ships on fire off the shoulder of Orion. I watched C-beams glitter in the dark near the Tannhäuser Gate. All those moments will be lost in time, like tears in rain ”

Rutger Hauer as *Roy Batty*s

Acknowledgements

I want to express my deepest gratitude to my supervisors, Prof. Dr. Miguel Areias and Prof. Dr. Jorge Barbosa, for their exceptional guidance, support, and encouragement throughout this research journey. Prof. Dr. Areias, as an Assistant Professor at the Faculty of Science of the University of Porto, and Prof. Dr. Barbosa, as an Associate Professor at the Faculty of Engineering of the University of Porto, have both provided invaluable insights that have significantly shaped the direction and quality of this thesis. Their extensive knowledge, constructive feedback, and unwavering commitment to my success have been pivotal in overcoming challenges and advancing this research.

I am also profoundly grateful to the entire SYCL community. The assistance, shared knowledge, and collective expertise of its members have significantly contributed to the progress and development of my work. Engaging with this community has been a rewarding experience, enhancing my understanding and application of SYCL technologies.

Finally, I wish to extend my heartfelt thanks to my family. Their unwavering support, understanding, and encouragement have been a constant source of strength and motivation throughout this demanding journey. Their belief in me has been a driving force, and their support has made this possible.

Abstract

Heterogeneous computing, which leverages the combined strengths of diverse processing units such as multicore CPUs and integrated GPUs, presents a promising approach to enhancing computational performance and efficiency. This thesis investigates the application of heterogeneous systems for two key computational tasks: Sparse Matrix-Vector Multiplication (SpMV) and Singular Value Decomposition (SVD). Through the development and analysis of various scheduling strategies, this work examines both the benefits and challenges of utilizing CPUs and integrated GPUs in tandem to accelerate these operations on mixed hardware architectures.

Experimental results demonstrate that integrated GPUs can provide significant performance improvements for SVD tasks, particularly for larger matrices. However, our analysis of the 50/50 scheduler reveals limitations in static workload partitioning, underscoring the need for adaptive scheduling mechanisms that adjust based on real-time performance metrics.

The findings of this thesis emphasize the potential of heterogeneous computing for accelerating scientific applications and offer insights into effective scheduling, workload distribution, and resource management.

Keywords: Heterogeneous Computing, Multicore CPU, Integrated GPU, SYCL, Sparse Matrix-Vector Multiplication, Singular Value Decomposition

Resumo

A computação heterogénea, que aproveita unidades de processamento diversificadas, como CPUs multicore e GPUs integradas, apresenta uma abordagem promissora para melhorar o desempenho e a eficiência computacional. Esta tese investiga a aplicação de sistemas heterogéneos para duas tarefas computacionais fundamentais: Multiplicação de Matrizes Esparsas por Vetores (SpMV) e Decomposição em Valores Singulares (SVD). Através do desenvolvimento e análise de várias estratégias de escalonamento, este trabalho examina os benefícios e desafios de utilizar CPUs e GPUs integradas em conjunto para acelerar estas operações em arquiteturas de hardware mistas.

Os resultados experimentais demonstram que as GPUs integradas podem proporcionar melhorias significativas de desempenho para tarefas de SVD, especialmente em matrizes de maior dimensão. No entanto, a nossa análise de um escalonador de carga 50/50 destaca limitações na partição estática de tarefas, sublinhando a necessidade de mecanismos de escalonamento adaptativos que se ajustem com base em métricas de desempenho em tempo real.

Os resultados desta tese enfatizam o potencial da computação heterogénea para acelerar aplicações científicas e oferecem insights sobre escalonamento eficaz, distribuição de carga e gestão de recursos.
Keywords: Heterogeneous Computing, Multicore CPU, Integrated GPU, SYCL, Sparse Matrix-Vector Multiplication, Singular Value Decomposition

Table of Contents

List of Figures

1. Introduction

Computing technology has evolved to embrace the concept of heterogeneous computing, where multiple types of processing units work together to enhance performance and efficiency. This approach capitalizes on the strengths of different computing resources, such as multicore CPUs and integrated GPUs (iGPUs), to tackle complex and demanding computational tasks.

This thesis is dedicated to investigating the potential and challenges of heterogeneous computing systems. The primary objective of this research is to explore how the combination of multicore CPUs and integrated GPUs can be leveraged to optimize performance and efficiency in computational tasks, specifically focusing on Sparse Matrix-Vector Multiplication (SpMV) and Singular Value Decomposition (SVD).

SpMV and SVD are fundamental operations in many scientific and engineering applications, and their efficient execution is critical for addressing large-scale problems. By examining the performance of these operations on heterogeneous platforms, this thesis aims to identify the advantages and limitations of using multicore CPUs and integrated GPUs in tandem.

The research is driven by the need to understand how different computational resources can be effectively utilized to improve overall system performance. It includes investigating various approaches for implementing SpMV and SVD, analyzing their performance across different hardware configurations, and exploring the benefits of running these operations concurrently on both CPUs and GPUs.

Through this exploration, the thesis seeks to contribute valuable insights into optimizing computational tasks on heterogeneous systems.

1.1. Thesis Structure

This thesis is organized in the following way:

Chapter 2, **State of the Art**, provides a comprehensive review of the existing literature and background relevant to the thesis. It covers fundamental concepts and technologies necessary for understanding the research. This chapter is divided into several key sections. Section 2.1 addresses sparse matrix representations, including Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) formats. Section 2.2 explores various

approaches to Sparse Matrix-Vector Multiplication, focusing on merge-based techniques. Section 2.3 introduces Singular Value Decomposition and its algorithms, such as Jacobi rotations and Lanczos iterations. In Section 2.4, the principles of heterogeneous computing are discussed, along with a comparison of dedicated and integrated GPUs. Section 2.5 provides an in-depth look at Intel's integrated GPU architecture, covering System on Chip (SOC) architecture, slices, subslices, and execution units. Finally, Section 2.6 compares different programming languages used for GPU programming, including CUDA, OpenCL, and SYCL.

Chapter 3, **Implementation Details**, delves into the specifics of the implementations used in the research. It provides detailed descriptions of the methods and techniques employed. Section 3.1 focuses on the implementation of the Compressed Sparse Row format. Section 3.2 describes the naive approach to Sparse Matrix-Vector Multiplication. Section 3.3 explains the implementation of the merge-based SpMV technique. Section 3.4 outlines the implementation of Jacobi rotations for SVD. Section 3.5 provides details on the implementation of Lanczos iterations. Finally, Section 3.6 outline our simple implementation of a naive scheduler for the Jacobi rotations.

The fourth chapter, **Experimental Results**, presents and analyzes the experiments' outcomes. Section 4.1 provides an analysis of the results for Sparse Matrix-Vector Multiplication. Section 4.2 examines the results related to Singular Value Decomposition. Section 4.3 discusses the results of running SVD kernels simultaneously on both the CPU and integrated GPU, exploring the performance of concurrent executions. Section 4.4 provides the results of our $50/50$ load scheduler for the Jacobi Kernel.

1.2. Main Contributions

This thesis's main contribution is exploring the performance and utility of heterogeneous computing systems, specifically through the implementation and evaluation of the Singular Value Decomposition (SVD) algorithm using *SYCL* on both CPU and integrated GPU hardware. The following are the specific contributions of this work:

- **Implementation of the SVD method in SYCL for execution on both CPU and GPU:** This thesis presents an implementation of the SVD algorithm in SYCL, taking advantage of SYCL's portability and ability to target heterogeneous hardware. The implementation includes two distinct approaches: one using Jacobi rotations and

another hybrid method that combines Lanczos spectral decomposition with Jacobi rotations for diagonalization.

- **Performance evaluation of the SVD:** The performance of the SYCL-based SVD implementation is evaluated across various hardware platforms, particularly the Intel® Core™ i5-6300U CPU and Intel® HD Graphics 520 integrated GPU. The study assesses how the algorithm performs with varying matrix sizes and investigates the distribution of computational workload between the CPU and GPU. A comprehensive runtime analysis highlights the relative advantages and limitations of each platform.

- **Evaluation of the integrated GPU as a computational accelerator:** This work evaluates the potential of the integrated GPU as a computational accelerator for the SVD algorithm. Performance comparisons between the CPU and integrated GPU are conducted to quantify the effectiveness of GPU acceleration. This evaluation provides insight into the feasibility of using integrated GPUs for high-performance computing tasks, such as SVD.

- **Assessment of the impact of heterogeneity on system performance:** The thesis explores how the simultaneous execution of independent SVD kernels on both the CPU and integrated GPU affects overall system performance. By leveraging both computational resources concurrently, this study examines the benefits and limitations of heterogeneous computing for large-scale linear algebra problems.

- **Design and analysis of a 50/50 workload scheduler for heterogeneous systems:** The thesis includes the design and evaluation of a 50/50 workload scheduler, which statically divides computational tasks equally between the CPU and integrated GPU. This analysis highlights the limitations of static workload partitioning and underscores the need for adaptive scheduling mechanisms that can adjust based on real-time performance metrics.

2. State of the Art

This chapter explores the concepts, data structures, and computational techniques that form the basis of sparse matrix operations, SVD methodologies and heterogeneous computing. Additionally, the chapter covers the architecture of integrated GPUs, particularly Intel's, and examines the merits and trade-offs of programming languages and frameworks like CUDA, OpenCL, and SYCL, which enable efficient development across varied platforms.

2.1. Sparse Matrices

A Sparse Matrix is a matrix where most of the elements are zero [1]. There is not a strict definition between sparse matrix dimensions and the number of nonzeros, but J. H. Wilkinson, a pioneer in numerical analysis and known for his contributions to matrix computations and numerical stability, gave an informal working definition of a sparse matrix as "any matrix with enough zeros that it pays to take advantage of them" [2].

Utilizing specific methods and data structures that take advantage of the sparse structure of the matrix is advantageous and frequently required when storing and working with sparse matrices on a computer. Sparse matrices are standard in machine learning [3]. Hence, specialized computers have been designed to handle them [4]. Standard dense-matrix structures and techniques are wasteful and slow for large sparse matrices because processing and memory resources are squandered on the zeros. Because sparse data is more compressible by nature, it requires much less storage space. Standard dense-matrix techniques are impractical for manipulating some huge sparse matrices [5].

In the following section, we will describe two common storage schemes for sparse matrices, the **Compressed Sparse Row** (CSR) and the **Compressed Sparse Column** (CSC or CCS) [5].

This thesis represents the row dimension, column dimension, and the number of nonzeros of sparse matrix A (2.1) by the variables m, n, and nnz, respectively.

This thesis represents the row dimension, column dimension, and the number of nonzeros of a sparse matrix by the variables m, n, and nnz, respectively. For the specific case shown in matrix A (2.1), we have $m = 5$ (rows), $n = 5$ (columns), and $nnz = 10$ (non-zero elements).

$$A = \begin{pmatrix} 0 & 0 & a_{02} & a_{03} & 0 \\ a_{10} & 0 & 0 & 0 & a_{14} \\ 0 & a_{21} & 0 & a_{23} & 0 \\ a_{30} & 0 & a_{32} & 0 & a_{34} \\ 0 & 0 & a_{42} & a_{43} & 0 \end{pmatrix} \tag{2.1}$$

Figure 2.1: Sparse matrix A.

2.1.1 Compressed Sparse Column (CSC)

This storage scheme for sparse matrices uses three arrays to store the sparse matrix A, where the nonzeros are stored column-by-column[1].

- *value*: for storing the nonzeros of A column-by-column.
- *rowind*: for storing the row index of each nonzero.
- *col ptr*: for storing the index of the first nonzero of each column in the value array.

value	a_{10}	a_{30}	a_{21}	a_{02}	a_{32}	a_{42}	a_{03}	a_{23}	a_{43}	a_{14}	a_{34}
rowind	1	3	2	0	3	4	0	2	4	1	3
colptr	0	2	3	6	9	11					

Table 2.1: CSC data structure for sparse matrix A.

The memory requirement for this scheme is given by the formula $8 \times \text{nnz} + 4 \times (\text{nnz} + n + 1)$ bytes (assuming an x86_64 architecture). The formula is deduced because the value array, containing the non-zero values of the matrix, holds double-precision floating-point numbers, so each non-zero element in the matrix will take up 8 bytes. Since nnz non-zero elements are in the matrix, the required memory will be $8 \times$ nnz bytes.

The `rowind` array is a store for the index of rows for a non-zero value. Every index is represented as an integer, which occupies 4 bytes. So, the memory required to store this array is $4 \times$ nnz bytes.

Finally, the $n + 1$ length array `colptr` is allocated, and it stores the index of the first non-zero value of each column, plus an extra index at the end for marking the end of the last column. The number of entries in `colptr` is $n + 1$, and each entry is an integer taking 4 bytes, and hence the memory required to store this would be $4 \times (n + 1)$ bytes.

2.1.2 Compressed sparse Row (CSR)

This is the most common storage scheme for sparse matrices [1], like the **CSC** format it uses three arrays to store the sparse matrix A:

- *value*: for storing the nonzeros of A row-by-row.
- *colind*: for storing the column index of each nonzero.
- *row ptr*: for storing the index of the first nonzero of each row in the value array.

value	a_{02}	a_{03}	a_{10}	a_{14}	a_{21}	a_{23}	a_{30}	a_{32}	a_{34}	a_{42}	a_{43}
colind	2	3	0	4	1	3	0	2	4	2	3
rowptr	0	2	4	6	9	11					

Table 2.2: CSR data structure for sparse matrix A.

The data structures to store the example matrix A under this storage scheme is shown in table 2.2.

This scheme requires $8 \times nnz + 4 \times (nnz + m + 1)$ bytes of memory. As it was for the CSC scheme, this is because of storing non-zero elements, the column index array, and row pointers. The term $8 \times nnz$ is the memory needed to store the non-zero entries, with each non-zero being a double-precision floating-point number, which takes 8 bytes. The total memory to store the values is $8 \times nnz$ bytes. The second term is $4 \times (nnz + m + 1)$, where the arrays *colind* and *rowptr* store integers. The *colind* array is of size nnz, storing the column index of the corresponding value in the other two arrays, i.e., in *val*. The *rowptr* array, of size $m + 1$, denotes the number of stored elements and their indices. The array *rowptr* has $m + 1$ entries, where m is the number of rows in the matrix and consists of the index of the first non-zero element for each row. There is one extra entry in *rowptr* at the end to signify the end of the last row. Since each entry in *rowptr* is also an integer, it requires 4 bytes per entry. Therefore, a total of $4 \times (nnz + m + 1)$ bytes of memory are required to store the *colind* and *rowptr* arrays.

2.2. Sparse Matrix-Vector Multiplication

The Sparse Matrix-Vector Multiplication (SpMV) is a core operation in scientific computing, finding applications across various domains such as linear algebra, graph analytics, machine learning, and other computational fields [6].

SpMV is essential in solving large-scale linear systems eigenvalue problems and is also the most time-consuming step in some iterative solvers like Conjugate Gradient. Therefore, the efficient implementation of SpMV is essential for enhancing the overall performance of these solvers.

Moreover, SpMV is a crucial operation in graph algorithms, especially in scenarios where the adjacency matrix exhibits sparsity. Notably, applications like PageRank, which underpin search engine algorithms, heavily rely on SpMV to compute the significance scores of nodes within a web graph [7].

Essentially, the SpMV operation encompasses multiplying a sparse matrix by a dense vector, resulting in another dense vector. When provided with a sparse matrix $A \in \mathbb{R}^{m \times n}$ and a dense vector $x \in \mathbb{R}^n$, the SpMV operation is represented as

$$y = A \times x$$

where $y \in \mathbb{R}^m$ signifies the resultant vector [8].

However, it is widely recognized that such a naive parallelization is often perceived as highly inefficient [9]. Making room for new algorithms, as explained in the section below, a Merge-Based SpMV.

2.2.1 Merge Based

The merge-based algorithm introduced by *NVIDIA* developers is a parallel method for doing Sparse Matrix-Vector Multiplication (SpMV) directly on the Compressed Sparse Row (CSR) matrix format (see Figure 2.2). The main objective of this algorithm is to ensure equal load distribution between processors in a way that will prevent the performance degradation arising from highly irregular row lengths in the matrix [10].

The main concept of the method is based on the ”merge-path” strategy that was originally developed to efficiently merge two sorted sequences. The idea is to view the parallel SpMV as a logical merging of two lists: the row descriptors and the natural numbers representing indices into the non-zero matrix values. The merge-path algorithm ensures that each processing element obtains an equal share of the work in breaking the entire problem into balanced parts. Two-dimensional searching is then used on the part each processor is to determine the range of rows and non-zero elements being processed [10].

The ability of the merge-based approach to distribute work among processing elements equally for matrices with irregular row lengths is very important. Unlike traditional methods of row-based parallelization that always end in imbalance because of varying lengths of rows, the merge-based strategy ensures there is no processor starvation by very long rows or very large numbers of zero-length rows. This is particularly effective for matrices whose structure is irregular and hard to predict [10].

In practice, the algorithm assigns work to each processor by using a 2D binary search on the diagonals of a conceptual grid. This grid represents the total amount of work to be done, where the rows are the x-axis and the non-zero elements are the y-axis. Each processor individually calculates the right SpMV, with no other processor input and in a way that guarantees work balancing among all processors as the work advances in this merge path in this grid [10].

It is not designed only for performance; it is also designed to be scalable. The algorithm can be applied recursively to partition the matrix across levels of parallelism and is quite suited to distributed environments in which local memories are fixed in size. Moreover, this approach uses no preprocessing or specialized data structures beyond the standard CSR format; thus, it helps minimize overhead and enhance portability across different systems [10].

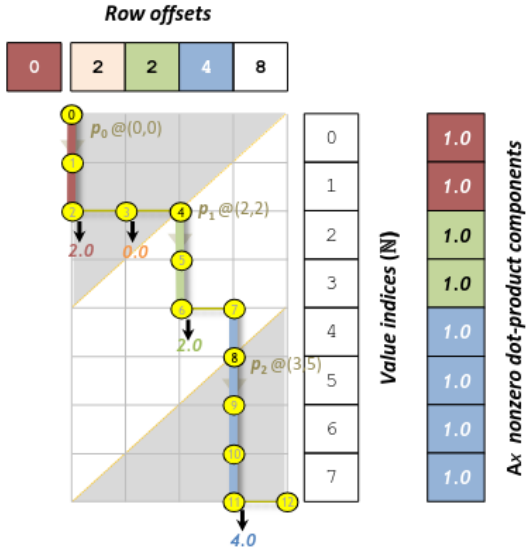

Figure 2.2: Merge-based `CsrMV` decomposition for three parallel threads. Inputs are the CSR matrix A from Figure 1 and the vector $x = [1.0, 1.0, 1.0, 1.0]$. The output is $y = [2.0, 0.0, 2.0, 4.0]$. Image from [10].

2.3. Singular Value Decomposition

Singular Value Decomposition (SVD) is a matrix factorization technique that plays a crucial role in various science and engineering applications. Its versatility is evident in addressing unconstrained linear least squares problems, matrix rank estimation, and canonical correlation analysis, among others. SVD is widely applied in fields such as information retrieval, seismic reflection tomography, and real-time signal processing [11].

The singular value decomposition of an $m \times n$ complex matrix $\mathbf{M}$ can be expressed as $\mathbf{M} = \mathbf{U\Sigma V}^*$, where $\mathbf{U}$ is an $m \times m$ complex unitary matrix, $\mathbf{\Sigma}$ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, $\mathbf{V}$ is an $n \times n$ complex unitary matrix, and $\mathbf{V}^*$ represents the conjugate transpose of $\mathbf{V}$. This decomposition exists for any complex matrix, while for real matrices, $\mathbf{U}$ and $\mathbf{V}$ are guaranteed to be real orthogonal matrices, denoted as $\mathbf{U\Sigma V}^T$ in such cases [11].

In the following sections, we will present algorithms for solving SVD, starting with two algorithms for dense matrices based on the Jacobi Rotations and on the Lanczos Method, an algorithm specifically designed for sparse matrices.

2.3.1 Jacobi Rotations

These techniques, known as Jacobi rotations or plane rotations, operate on the matrix without converting it into a different format, allowing this method to obtain excellent accuracy. Jacobi introduced this method in 1848 to solve the symmetric eigenvalue problem [12] by diagonalizing the matrix through a series of plane rotations.

$$A_{(0)} = A, \quad A_{(k+1)} = J_k^T(i,j,\theta)A_{(k)}J_k, \quad A_{(k)} \to \Lambda \text{ as } k \to \infty.$$

Each plane rotation, called a Jacobi rotation, $J_k = J_k(i,j,\theta)$ is an orthogonal matrix that differs from the identity only in rows and columns i and j:

$$J(i,j,\theta) = \begin{bmatrix} I & & & \\ & c & s & \\ & -s & c & \\ & & & I \end{bmatrix},$$

where $c = \cos\theta$ and $s = \sin\theta$. The angle θ is chosen in order to eliminate the pair a_{ij}, a_{ji} by applying $J(i,j,\theta)$ on the left and right side of the matrix A, which can be viewed as a simple 2×2 eigenvalue problem,

$$J_k^T \hat{A}_k J_k = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} d_{ii} & 0 \\ 0 & d_{jj} \end{bmatrix} = \hat{A}_{(k+1)},$$

where the notation $\hat{A}$ is the 2×2 submatrix of matrix A. Subsequent eliminations will fill in the eliminated entry, but at each step, the norm of off-diagonal elements,

$$\begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix}$$

$$\mathsf{off}(A) = \|A - \mathsf{diag}(A)\|_F = \left(\sum_{i \neq j} a_{ij}^2 \right)^{1/2},$$

is reduced until the matrix converges to diagonal form, Λ, revealing the eigenvalues. Accumulating the plane rotations, $V = J_{(0)} J_{(1)} \ldots$, gives us the eigenvectors.

This algorithm converges after several sweeps, generally 5–10. Wilkinson [13] proved that the convergence is quadratic, meaning that the rate at which the error decreases as the method proceeds. More precisely, after each sweep (or iteration), the error, which could be measured as the norm of the off-diagonal elements, decreases in the proportionality of the square of the previous error. Mathematically speaking, if the error at the k-th iteration is denoted by e_k, then at the next iteration, the error is approximately proportional to $(e_k)^2$. This fast drop implies that once an algorithm gets near a solution, the remainder of the error goes very rapidly to zero; hence, there is faster convergence towards the solution.

2.3.1.1 Two-Sided Jacobi SVD

Jacobi's eigenvalue method was generalized to the SVD of a general, nonsymmetric matrix in two different ways. The first way is the two-sided method created by Kogbetliantz [14], which applies two different plane rotations, $J(i,j,\theta)$ on the left of the matrix A and $K(i,j,\phi)$ on the right of A, to eliminate the a_{ij} and a_{ji} entries. Sweeps are done over the off-diagonal entries until the norm of off-diagonal entries is below a specified tolerance

(usually if precision is a must, then machine epsilon is used as tolerance), revealing the singular values, Σ, via the iteration:

$$A_{(0)} = A, \quad A_{(k+1)} = J_{(k)}^T A_{(k)} K_{(k)}, \quad A_{(k)} \to \Sigma \text{ as } k \to \infty.$$

The left singular vectors are obtained by collecting the left rotations, $U = J_{(0)} J_{(1)} \ldots$, while the right singular vectors are obtained by accumulating the right rotations, $V = K_{(0)} K_{(1)} \ldots$.

Calculating $J(i, j, \theta)$ and $K(i, j, \phi)$ can be viewed once again as solving a 2×2 SVD problem as shown in equation 2.2.

$$J_{(k)}^T \hat{A}_{(k)} K_{(k)} = \begin{bmatrix} c_J & s_J \\ -s_J & c_J \end{bmatrix} \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c_K & s_K \\ -s_K & c_K \end{bmatrix} = \begin{bmatrix} d_{ii} & 0 \\ 0 & d_{jj} \end{bmatrix} = \hat{A}_{(k+1)}. \tag{2.2}$$

2.3.1.2 One-Sided Jacobi SVD

Hestenes, a mathematician known for his contributions to numerical analysis and optimization, introduced a method [15] that uses plane rotations applied solely to the right side of A in order to orthogonalize its columns. This approach implicitly carries out the two-sided Jacobi eigenvalue method on $A^T A$. The product $U\Sigma$, representing the left singular vectors scaled by the singular values, is where the columns of A converge.

$$A_{(0)} = A, \quad A_{(k+1)} = A_{(k)} J_{(k)}, \quad A_{(k)} \to U\Sigma \text{ as } k \to \infty.$$

Implicitly, this indicates that $A_{(k)}^T A_{(k)} \to \Sigma^2$. $V = J_{(0)} J_{(1)} \ldots$, the sum of the rotations, yields the correct singular vectors.

By solving the 2×2 eigenvalue issue, the rotations are found akin to the Jacobi eigenvalue approach.

$$J_{(k)}^T \begin{bmatrix} b_{ii} & b_{ij} \\ b_{ij} & b_{jj} \end{bmatrix} J_{(k)} = \begin{bmatrix} d_{ii} & 0 \\ 0 & d_{jj} \end{bmatrix},$$

where a_i is the i-th column of $A_{(k)}$, and $b_{ij} = a_i^T a_j$. It computes the matrix $B = A^T A$ during a sweep, but instead of applying J directly to $A^T A$, it applies it to A, avoiding the numerical instabilities related to $A^T A$. This is the one-sided Jacobi approach described in [16].

When $|b_{ij}| < \epsilon\sqrt{b_{ii}b_{jj}}$, it indicates that columns a_i and a_j are already numerically orthogonal and so skips rotations. When every rotation in a sweep is omitted, it converges. To achieve high relative accuracy, convergence must be verified using this formula. When skipping rotations, fewer operations are required since the b_{ii} column norms can be stored instead of recomputed for each pair. The last sweep, which does no significant work other than checking for convergence, requires roughly n^3 flops to compute b_{ij} terms.

2.3.2 Lanczos Iterations

One of the widely recognized approaches in addressing large, sparse, symmetric eigenproblems stems from a technique developed by Lanczos in 1950 [17]. This technique constructs a sequence of tridiagonal matrices, denoted as T_j, where each matrix is of dimension $j \times j$. The key feature of this method is that the eigenvalues of these tridiagonal matrices provide increasingly accurate approximations of the eigenvalues of the original matrix as j increases. The Lanczos algorithm is particularly powerful when applied to symmetric matrices, as it can effectively reduce the dimensionality of the problem while preserving key spectral properties [17].

The main idea behind the Lanczos method is to transform the original matrix A into a smaller tridiagonal matrix T_j whose eigenvalues approximate the eigenvalues of A. This method is particularly useful for large, sparse matrices, where computing the full eigenvalue decomposition can be computationally expensive. The vectors generated during the Lanczos process form an orthogonal basis, and the tridiagonal matrix T_j can be analyzed to obtain approximations of the singular values and vectors [17].

The pseudo-code for the Lanczos algorithm is presented in Algorithm 1.

Algorithm 1 Lanczos Algorithm

$q_1 \leftarrow$ Random vector with norm 1
$q_0 \leftarrow 0$
$\beta_1 \leftarrow 0$
for $j \leftarrow 1$ **to** m **do**
    $w_j \leftarrow A \cdot q_j - \beta_j \cdot q_{j-1}$
    $\alpha_j \leftarrow w_j \cdot q_j$
    $w_j \leftarrow w_j - \alpha_j \cdot q_j$
    $\beta_{j+1} \leftarrow \|w_j\|$
    $q_{j+1} \leftarrow w_j / \beta_{j+1}$
    Compute eigenvalues, eigenvectors, and error bounds of T_j
end for

The key steps of the Lanczos process involve iteratively constructing the tridiagonal matrix T_j and calculating the eigenvalues of this matrix to approximate the singular values of the original matrix. This method is well-suited for applications requiring only the largest singular values or where computational efficiency is crucial due to the sparsity of the matrix [18].

2.4. Heterogeneous Computing

2.4.1 Concepts

Heterogeneous computing is a computer system that uses more than one kind of processing device, most often different processors such as general-purpose CPUs, GPUs, and FPGAs. This technique helps optimize the overall sustainability and functionality of the system. Because of its high general-purpose processing prowess, heterogeneous computing is becoming indispensable in various fields.

A fundamental concept in heterogeneous computing is the idea of ”jobs.” A job refers to a discrete unit of work or task that a processor performs, such as executing a specific algorithm, rendering a graphical frame, or processing a dataset.

The key benefit of heterogeneous computing is higher performance, as jobs can be allocated to an appropriate processor that is better suited for the task, which, in most cases, outperforms homogeneous systems. It is also energy efficient since processors can optimize their tasks, reducing overall power consumption. Flexibility and scalability become the most crucial ones, enabling a more configurable, customizable architecture when technological needs change.

In general, using CPUs combined with other processors like GPUs has speeded up computation immensely, contributing a lot to scientific research and simulations. The same happens with the gaming and graphics-rendering sector: though the GPU takes care of the rendering, the CPU processes the game logic and physics. Besides, the extensive data set management carried out by data centers benefits from heterogeneous architectures, which distribute workloads efficiently over processors of different types.

This thesis will focus on heterogeneous systems combining CPUs and GPUs.

2.4.2 Dedicate GPU vs Integrated GPU

2.4.2.1 Dedicated GPUs (dGPUs)

Dedicated GPUs are discrete hardware, each with its memory. They generally have more powerful specifications than integrated GPUs: high clock speeds, more processing cores, and a large memory bandwidth. This makes them excellent not only in the performance of graphics tasks but also in the rendering of gaming, 3D modeling, and professional video editing, as well as compute-intensive applications like deep learning and scientific simulations [19].

One of the significant advantages of dGPUs lies in the dedicated memory, VRAM (Video RAM), used only by the GPU. This allows fast access to large datasets and textures, crucial in graphics-intensive applications. In addition, they are independent components; thus, dGPUs will not fight with the CPU in terms of memory resources like integrated GPUs, leading to better overall system performance.

However, latencies are introduced while transferring data between the CPU and dGPU, especially when dealing with massive datasets. This data is transferred over the PCIe bus, which is actually very fast but still not as fast as the integrated GPUs.

2.4.2.2 Integrated GPUs (iGPUs)

Integrated GPUs, on the other hand, are built directly into the CPU and share the system's memory with the CPU. While usually less potent than dedicated GPUs, they offer many advantages regarding power efficiency, cost, and space within essential systems such as laptops and compact desktops [19].

The other advantage of iGPUs is a lower latency in data movement from CPU to GPU because they share memory, which avoids the need for communication over a separate bus like PCIe, which is commonly used by dedicated GPUs. However, it is important to note that although there is no separate bus like PCIe involved, there is still communication between the CPU and the memory (RAM), which is connected via memory controllers. This might lead to data processing being done at a higher speed wherever communication between CPU and GPU is abundant or when the data size is small enough to fit snugly in shared memory.

However, iGPUs' shared memory architecture also means that the GPU competes with the CPU for memory resources, which can result in reduced performance for both the CPU

and GPU when under heavy load. For instance, during simultaneous intensive tasks such as running a complex simulation on the CPU while rendering graphics on the iGPU, both the CPU and iGPU will compete for access to the same memory controller. In this scenario, operations such as load/store instructions executed by the CPU may overlap with those from the iGPU, creating contention for the memory controller and leading to latency in data access. This competition is particularly noticeable in high-memory bandwidth tasks where both components are trying to perform frequent memory reads and writes.

In addition, an iGPU usually has a much lower clock speed and fewer cores than a dGPU; thus, an iGPU is less suitable for intensive tasks, such as parallel processing or high-end graphic rendering.

2.5. Intel Integrated GPU Architecture

This section focuses on Intel's integrated GPU architectures, specifically Gen9.5, the architecture used in this work study, to explore the potential of iGPUs in more detail.

2.5.1 Intel SOC Architecture

The Intel Gen9.5 graphics architecture was introduced on August 30, 2016 [20], and represents an evolution from its predecessor, Gen9, with various incremental upgrades. The most notable improvement is the shift to a 14-nanometer fabrication process, resulting in reduced power consumption. This architecture comprises three primary components: the **Slice**, the **Unslice**, and the **Display components** [21–23].

A comprehensive overview of the system-on-chip (SoC) architecture is depicted in 2.3, illustrating the interconnectedness of the CPU cores and the GPU. The Graphics Technology Interface (GTI) serves as the link between the GPU and the SoC Ring Interconnect. This ring network topology facilitates connections among all agents on the die, using a bi-directional 32-byte wide data bus with separate channels for requests, snooping, and acknowledgments. The connected agents include the GPU, the CPU cores, the Last Level Cache (LLC), and the System Agent, housing various components such as the Dynamic Random Access Memory (DRAM) Management Unit and I/O controllers like PCIe [21, 23].

Moreover, the GTI houses global memory atomics hardware shared by both the GPU and CPU cores and manages power for the GPU to accommodate different clock domains. The bus connecting the GTI to the GPU supports 64 B/cycle read and write operations,

with an optional configuration that reduces write throughput to 32 B/cycles for low-power scenarios [21, 23].

Focusing on the GPU, Figure 2.4 presents a comprehensive overview of the three primary components of the GPU and their interplay. The predominant focus of this thesis will be the exploration of the Slice components, where all GPGPU (General-purpose computing on graphics processing units) computations occur. The Unslice, responsible for the 3D and Media pipelines which are beyond the thesis scope, will be briefly discussed due to their interactions with the Slice and their relevance in the counter architecture. The Unslice houses fixed-function media and geometry hardware, providing access to the LLC and, consequently, to the DRAM through the GTI. The Unslice incorporates the 3D and Media Pipelines, executing 3D shaders and media kernels, respectively. Within the Unslice, the Command Streamer (CS) oversees the execution of the 3D and Media pipelines, managing the context switch between the two and forwarding command streams from the CPU to the pipelines' different stages. The Command Streamer also administers the Unified Return Buffer (URB), a general-purpose buffer in the L3 cache facilitating data transfer between threads or between threads and fixed function units. While integral to the 3D pipeline, the URB is not chiefly used for GPGPU computing but may be utilized by certain fixed function pipeline stages to request general-purpose computation from the Execution Units (EU) in the Slice. To facilitate this, communication occurs with the Global Thread Dispatcher (GTD), providing URB handles and other requisite data types [21, 23].

Fixed-function pipeline stages often request general-purpose computation from the Execution Units (EUs) in the Slice, facilitated by the Global Thread Dispatcher (GTD), which loads URB entries into the General Register File (GRF) in the EUs. The Command Streamer parses commands from the CPU, routing workloads to appropriate GPU units, whether they are 3D tasks from graphics libraries like DirectX or OpenGL, media workloads for video decoding, or GPGPU kernels from OpenCL or other environments. The GTD operates in two modes: maximizing occupancy for workloads without barriers or Shared Local Memory (SLM) access and optimizing thread group arrangements for those with such requirements [21, 23].

The Media pipeline manages media workloads and performs general-purpose computing. Within the Media pipeline's Video Front End (VFE), the Thread Spawner initializes threads for GPGPU APIs like OpenCL. The 3D pipeline consists of eight stages, each

serving specific functions. It commences with the Vertex Fetch (VF), responsible for formatting vertex data and recording results to Vertex URB Entries (VUE). Following this, the pipeline progresses through the Vertex Shader (VS), Hull Shader for tessellation, Tessellation Engine (TE), Domain Shader (DS), Geometry Shader, Stream Output Logic (SOL), Clipper (CLIP), Strip/Fan (SF), and Windower/Masker (WM). The rasterization process within the Windower/Masker (WM) maps pixels to primitives, ultimately presenting them on the screen [20, 23].

As previously mentioned, the Slice contains general-purpose computation units for calculations required by other pipelines and GPGPU kernels. It is divided into sub-slices, each housing multiple execution units (EUs) for computations. The Unslice allocates EU usage for 3D or Media pipeline computations via the GTD, which assigns computations to hardware threads and allocates EU register space. Depending on the SKU, an Intel iGPU may feature various configurations of Slices, Subslices, and EUs. In the Intel Gen9.5 microarchitecture, there are five distinct SKU configurations: GT1 (1 slice, two subslices, 6 EUs/subslice), GT1.5 (1 slice, three subslices, 6 EUs/subslice), GT2 (1 slice, three subslices, 8 EUs/subslice), and GT3 (2 slices, three subslices, 8 EUs/subslice) [20, 23].

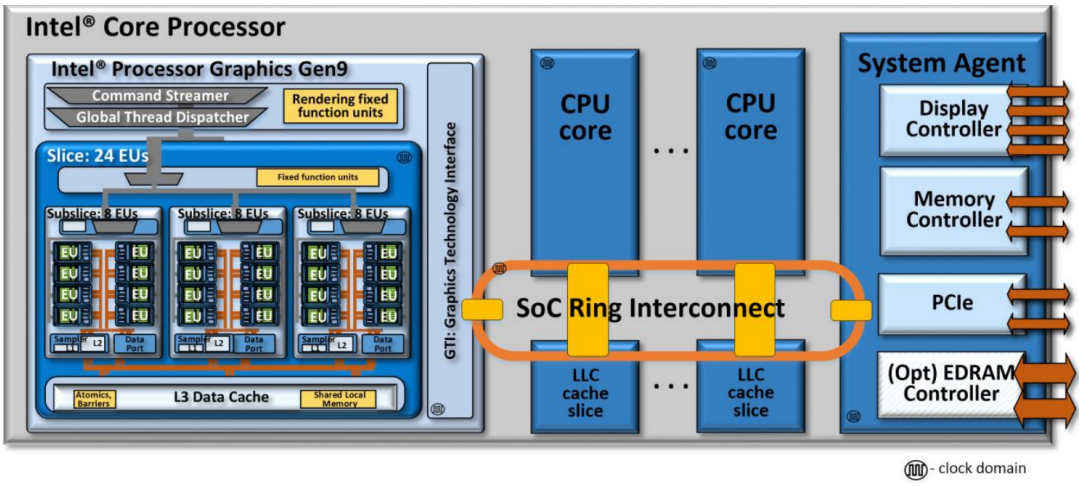

Figure 2.3: Intel Gen9 SoC Architecture. Image from [21].

2.5.2 Slices Architecture

The architecture depicted in the Slices (Figure 2.5) comprises subslices, banked L3 cache, and Shared Local Memory (SLM). Each slice also encompasses fixed functions for atomic operations, 3D, and media processing functions.

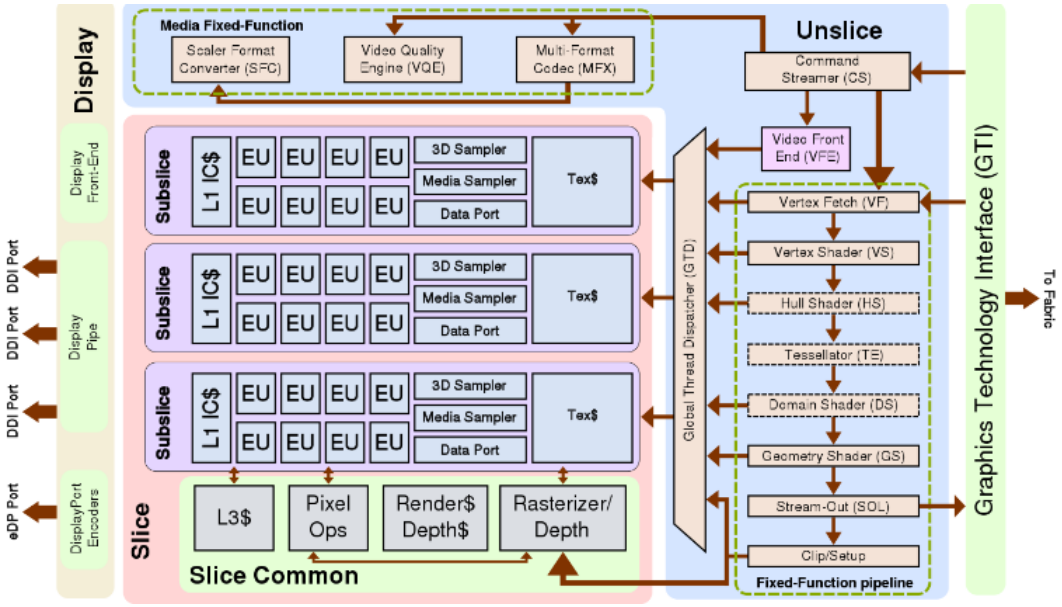

Figure 2.4: Intel Gen9 GPU Layout. Image from [20].

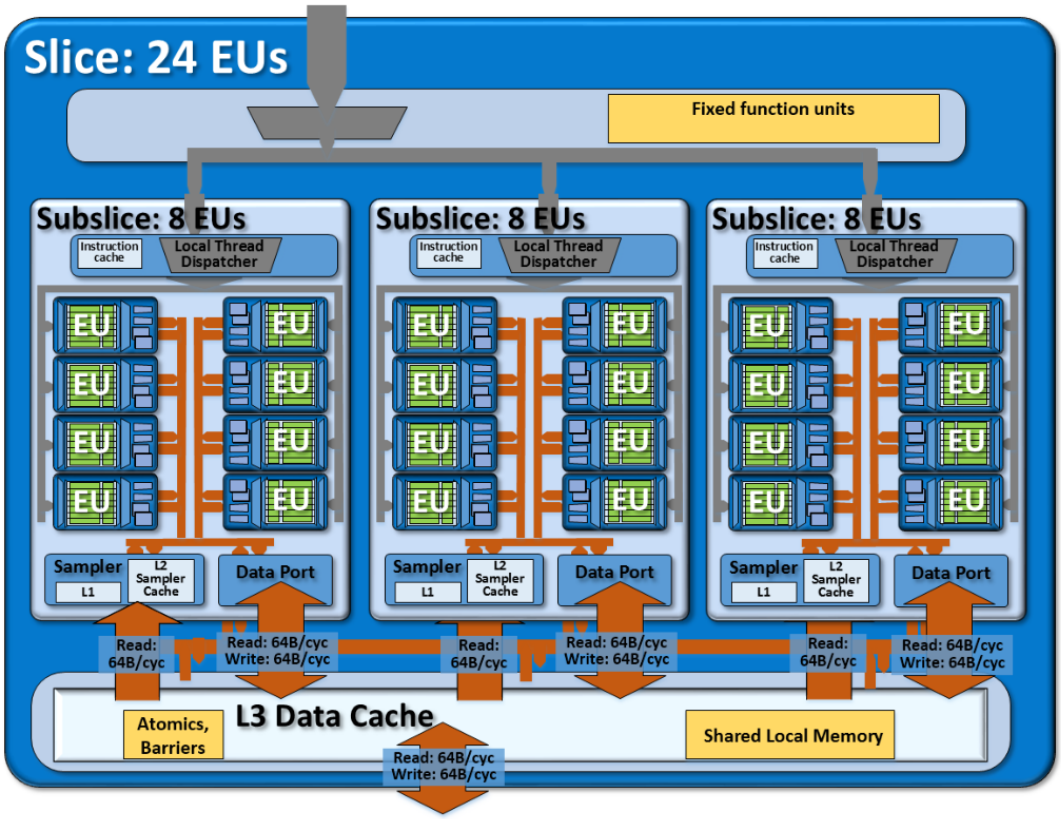

Figure 2.5: Intel Gen9 Slice. Image from [21].

Distinct from the CPU's L3 Cache, the L3 data cache in the GPU, referred to as the LLC when discussing the GPU, operates as an L4 cache in the context of the Sampler caches. It has a capacity of 768 KB per Slice with 64-byte cache lines. Each subslice has its own L3 partition, which collectively functions as a single cache for the Slice. In configurations with multiple slices, the L3 partitions from each subslice across all slices combine to form a unified GPU-wide L3 cache. The Samplers and Data Port units are equipped with individual interfaces to the L3 cache, with support for read and write bandwidths of 64 bytes per cycle. This bandwidth scales linearly with the number of subslices in the GPU. For instance, in the GT2 configuration with a single slice, the total theoretical bandwidth equates to 64 bytes/cycle × 3 subslices, resulting in 192 bytes/cycle. However, achieving this upper bound necessitates careful workload partitioning across the slices. Data transfers to and from the L3 occur in 64-byte packets (cache lines). Furthermore, the OpenCL Just-In-Time (JIT) compiler optimizes data fetches by grouping them to minimize wasted bandwidth [23].

The SLM, found within the L3 cache, serves as programmer-managed scratchpad memory for sharing data across different EUs within the same subslice. Workloads such as 2D image processing and Finite-Difference Time-Domain (FDTD) computation benefit from data sharing between threads, particularly those with constraints at their borders and are not massively parallel. Each subslice has access to 64 KB of SLM, which, despite being situated in the L3 cache, does not have lower latency. Nevertheless, it is more highly banked, allowing total throughput for non-64-byte-aligned access patterns or non-contiguously adjacent data in memory. Unlike the L3 cache, SLM is not coherent with other memory structures [21, 23].

2.5.3 Subslices Architecture

The subslice encompasses multiple Execution Units (EUs) that are accountable for executing computations. The number of EUs housed in a subslice is contingent on the specific product configuration. For instance, subslices within GT1 and GT1.5 configurations are each equipped with 6 EUs, whereas those within GT2, GT3, and GT4 configurations consist of 8 EUs [20]. Alongside the EUs, a subslice is inclusive of a Local Thread Dispatcher (LTD), an Instruction Cache, a Data Port unit, and a Sampler unit [21].

The LTD is tasked with distributing the workload among the EUs contained within the subslice. It collaborates with the Global Thread Dispatcher (GTD) in the Unslice to ensure

optimal thread coverage. Although the LTD does not engage in direct communication with the GTD, it endeavors to equitably distribute the threads assigned by the GTD to the EUs [21].

The Sampler, comprised of L1 and L2 read-only caches, is specifically dedicated to texture and image surfaces. It incorporates fixed-function hardware for address and image coordinate conversion, various clamping modes (mirror, wrap, border, and clamp), and sampling filtering modes (including point, bilinear, trilinear, and anisotropic). Primarily, these operations are reserved for 3D workloads and are not extensively utilized in GPGPU tasks. The Sampler sustains a read throughput of 64 bytes per cycle from the L3 cache [21, 23].

The Data Port unit interfaces with the L3 fabric within the Slice and manages memory load and store requests. It facilitates read and write throughputs of 64 bytes per cycle, mandating that all memory requests from the EUs must traverse through the Data Port unit [21, 23].

2.5.4 Integrated GPU Execution Unit Architecture

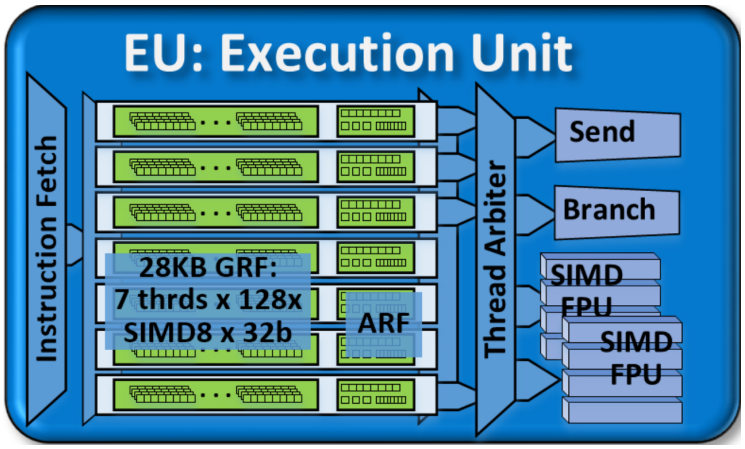

Figure 2.6: Intel Gen9 Execution Unit. Image from [21].

The Execution Units (EUs) are equipped with a register file comprising 128 general-purpose 256-bit registers per thread, collectively known as the General Register File (GRF). Additionally, each EU includes a set of architecture-specific registers, referred to as the Architectural Register File (ARF), primarily utilized for resource management, such as maintaining the instruction pointer per thread and managing dependencies. These registers are mainly leveraged by the hardware and driver [23].

Each EU is composed of four functional units: the Send unit, the Branch unit, and two Floating Point Units (FPUs). This configuration enables an EU to co-issue up to four instructions per cycle, provided they originate from different threads. Each EU supports up to seven hardware threads to ensure sufficient workload for maintaining this throughput. Consequently, the total GRF capacity for a single EU amounts to 128 x 256 bits / 8 / 1024 * 7 threads = 28 KB, with 4 KB allocated per thread [23].

Each send instruction includes a pointer to the message payload, which the Send unit uses to format messages correctly for the Data Port unit in the subslice. The Data Port unit then forwards the memory access requests and returns the data to the GRF (in case of a load) [23].

The branch unit manages branch instructions, which is crucial to optimizing performance. The EU's Instruction Set Architecture (ISA) includes mechanisms for branch prediction and mitigating thread divergence, a common performance issue where threads follow different execution paths based on comparison outcomes. If all threads follow the same route, no branching overhead is incurred. However, if they diverge, all affected threads execute both paths to prevent pipeline stalling, introducing computational overhead. This scenario often arises with conditional instructions and loop control [21, 23].

The branch unit is responsible for managing branch instructions, playing a crucial role in optimizing performance. The Instruction Set Architecture (ISA) of the EU incorporates features for branch prediction and addressing thread divergence, a common performance concern where threads take different execution paths based on comparison outcomes. When all threads follow the same route, no branching overhead is experienced. However, if thread paths diverge, all affected threads execute both paths to prevent pipeline stalling, leading to increased computational overhead. This situation often arises with conditional instructions and loop control [21, 23].

Each EU contains two FPUs, which handle single and half-precision floating-point and integer computations. Additionally, one FPU is capable of performing double-precision floating-point computations and transcendental mathematical operations. Although the EU's ISA supports logical instructions of various 32-bit data types, the FPUs are physically 4-wide, allowing for four simultaneous executions of single-precision floating-point operations per FPU. These units are fully pipelined for wider vector types, enabling them to achieve maximum throughput regardless of the specified SIMD width by reusing the

same execution units. Each FPU is capable of executing one single-precision multiply-add (MAD) operation per cycle. Therefore, the EU can achieve a theoretical throughput of 16 single-precision floating-point operations per cycle: 2 (MAD) × 2 FPUs × SIMD-4 = 16 FLOP/cycle. However, since only one FPU supports double precision and double-precision operations offer half the throughput of single-precision, the double-precision throughput is reduced fourfold, resulting in 4 FLOP/cycle for MAD operations [21, 23].

2.6. Heterogeneous Computing Frameworks

This section overviews current standards/computing frameworks that focus on heterogeneous systems. Starting with *CUDA*, passing by *OpenCL*, and ending with *SYCL*, the language in which this thesis work is developed.

2.6.1 Cuda

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface model created in 2006 by NVIDIA to expand the possibility of programming GPUs for graphics and shader purposes and general computing [24].

To enable the programming of GPUs, NVIDIA initially contemplated several possibilities, including designing brand new languages from scratch or designing OpenGL extensions for this purpose. It was decided that using a new language would hinder GPU programming, so NVIDIA adopted the C language as the foundation for CUDA. Over time, *CUDA* has incorporated support for many modern C++ features and has also added compatibility with other programming languages, such as Fortran [25].

The developers themselves define the code that will be executed on the GPU through a keyworded function known as kernel. In the kernels, developers can define the type of parallelization strategy they need and the local memory required for that kernel. This means all control regarding designing the code and its utilization goes directly into the hands of the developers. At the same time, the compiler compiles the code for the appropriate machine using a certain specific set of declaration specifiers from NVIDIA.

CUDA also allows developers to select and access individual threads exactly, blocks, or grids executing via constructs such as threadId.x, threadId.y, or blockId.x in conjunction with similar keywords. Even though the evolution of *CUDA* has made it more CPP and less C-like, the latter model has been maintained in memory handling [26].

The most compelling benefit of the proprietary approach to heterogeneous programming is that it is performance-optimized based on hardware-software. For this reason, it would limit code portability for other platforms and make it much harder for consumers to add new functionalities.

2.6.2 OpenCL

OpenCL [27] is an open standard for cross-platform, parallel programming that targets various types of computing devices, including accelerators such as GPUs and FPGAs, as well as traditional processors. Managed by the *Khronos Group*, it ensures that company implementations conform to the standard through certification.

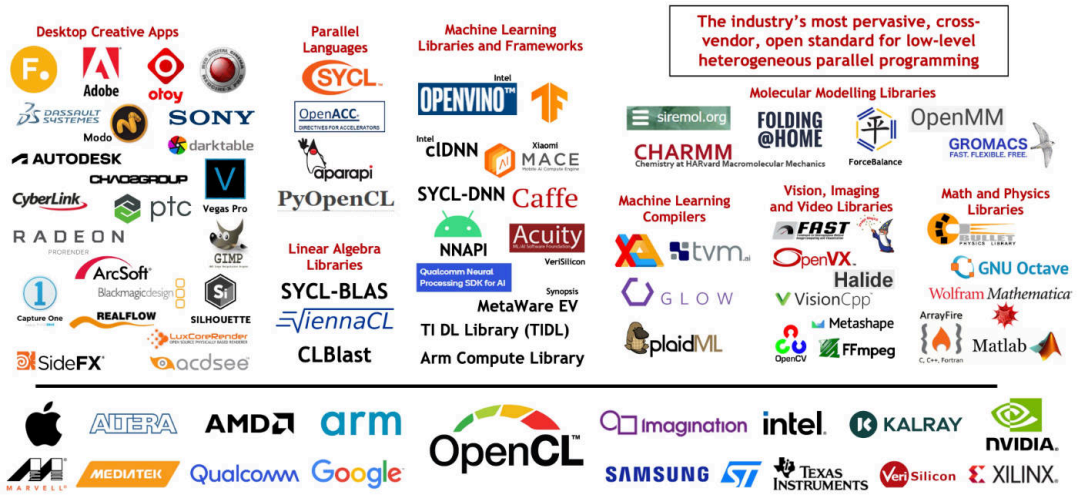

Figure 2.7: *OpenCL* users and major contributors. Image from [27]

As shown in figure 2.7, *OpenCL* is used by many companies across different fields and is now a viable alternative to CUDA for GPGPU programming. Unlike CUDA, *OpenCL* is not controlled by a single company and supports diverse hardware.

OpenCL's hardware-agnostic nature results in a parallelism approach that differs from CUDA's. It does not influence hardware evolution but offers similar features to NVIDIA's technology, based on the collective needs of companies in the *Khronos Group*.

The standard is based on the C99 version of the C language [28], and the fact that it is low-level requires developers to manage many details themselves. To address this, since *OpenCL* 2.0, code can also be written in CPP. Developers write code for accelerators in specific functions called kernels, which are compiled and sent to the appropriate hardware. This method, shared with CUDA, allows for a fine-grained division of the workload managed by programmers. *OpenCL* provides two technical APIs: the platform API layer,

which enables a program to discover available parallel processors and adapt the code accordingly, and the Runtime API, which handles kernel compilation and communication between the host and accelerator [28].

The *OpenCL* standard has both advantages and limitations. Its primary advantage is portability, enabling good performance across a wide range of hardware, making it a valuable option for companies needing to ship software across diverse systems. Multiple simultaneous standards can result in compatibility issues, as seen when NVIDIA froze its *OpenCL* support at version 1.2, affecting code portability and hindering language evolution. Recently, NVIDIA announced support for *OpenCL* 3.0, improving its standing compared to AMD.

Another limitation is the initial choice of C99 as the kernel language standard. While beneficial initially, it hindered the evolution to higher-level languages like CPP, which offers more flexibility. Although CPP has been implemented since *OpenCL* 2.2, fragmentation issues prevent it from becoming the standard.

2.6.3 SYCL

The latest standard in heterogeneous computing is *SYCL*, which is managed, sponsored, and disseminated by the Khronos Group [29]. *SYCL* builds on the legacy of *OpenCL* by offering a heterogeneous language capable of running across diverse platforms with performance comparable to natively developed languages.

As a standard, *SYCL* facilitates the creation of code for various heterogeneous systems, from FPGAs to GPGPUs, using a unified syntax and compiler. Different companies have developed their compilers despite being a standard, although some have collaborated to create a unified compiler. Most of these compilers are based on LLVM Clang, thereby supporting most features provided by LLVM [30].

Explaining the functioning of *SYCL* can be challenging due to its implementation variability across different compilers and companies. As of late 2023, only three compilers are particularly relevant for GPGPU programming: DPC++ by Intel, hipSYCL by the University of Hamburg, and ComputeCpp by Codeplay. Codeplay has joined forces with Intel to enhance *SYCL*'s performance on NVIDIA GPUs to match *CUDA* levels. DPC++ also supports AMD GPUs using the *HIP* stack and SPIR-V as an intermediate representation. Intel's significant investment in DPC++ is due to its central role in their HPC platform and its goal to integrate the *SYCL* compiler into the Clang standard compiler [29].

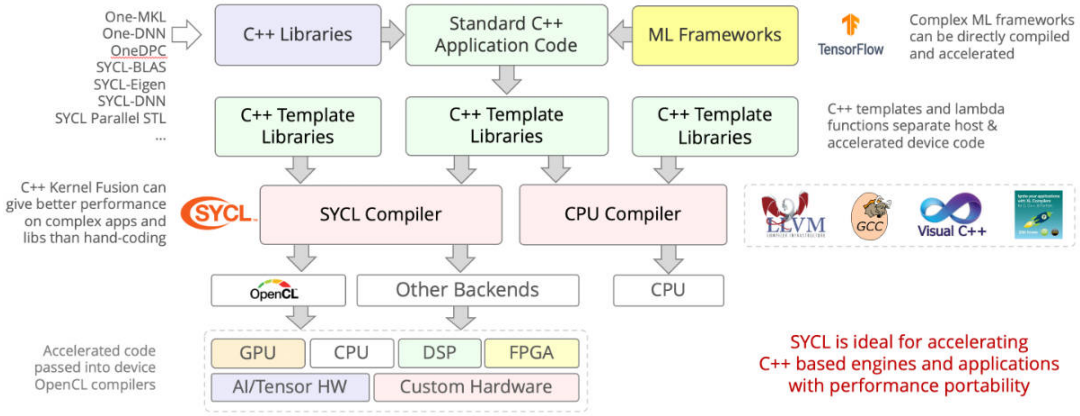

Figure 2.8: *SYCL* compilers implementations. Image from [29]

Despite the differences in compiler implementations, SYCL's general approach remains consistent. Similar to *OpenCL*, it aims to lower the programming effort required for heterogeneous platforms and enhance code portability. However, *SYCL* is independent of any specific company, eliminating the need for additional proprietary code, although it can still utilize existing toolkits and compilers provided by these companies.

SYCL handles the implementation and differentiation of accelerator-specific code through kernels, akin to *CUDA* and *OpenCL* [31]. This can be achieved by writing a lambda function or a class functor. The selection of accelerators occurs at runtime, using available hardware on the machine. Compilation and execution are decoupled, with the framework optimizing performance for the given hardware. While knowing the specific GPU at compile-time is beneficial for creating the correct assembly, it is not required during code development.

One significant advantage of *SYCL* is that the generated code is indistinguishable from that of proprietary compilers, allowing developers to use existing performance analysis tools seamlessly. This compatibility provides a significant advantage by offering a wide range of tools for code verification without slowing down the adoption process due to standard dependency.

2.6.4 Comparison

There are already some papers focusing on comparing these languages' strengths and weaknesses [32–34], with some concluding that the performance of *SYCL* and CUDA are similar.

Intel, one of the biggest backers of *SYCL*, also came to the same conclusion, stating that their results show that *SYCL* is highly performant on Nvidia and AMD devices and performs comparably to native CUDA or HIP code for diverse workloads [35].

In the comparison between *SYCL* and CUDA on an Nvidia system across ten workloads, *SYCL* either outperformed or matched CUDA in six workloads, including computationally intensive tasks such as the Sobel filter, Reverse Time Migration, and Bitcracker. The remaining four workloads showed only minor performance differences between *SYCL* and CUDA, with CUDA slightly outperforming *SYCL* in specific scenarios. However, these differences were marginal, suggesting that *SYCL* provides near-equivalent performance to CUDA in many real-world applications.

For AMD GPUs, the comparison between *SYCL* and HIP on an AMD system across nine workloads revealed that *SYCL* and HIP delivered comparable performance in most cases. Notably, *SYCL* achieved superior performance in workloads such as the Easywave and Ethminer benchmarks, which showed up to 90% better performance in some cases. On the other hand, HIP performed marginally better in workloads such as the Sobel filter and the *SYCL* HP-Linpack test, but again, the performance difference was not significant.

When comparing *SYCL* to CUDA and HIP, the slight differences in benchmark results can be attributed to variations in the maturity and optimization of the respective compiler and runtime toolchains. While *SYCL* demonstrates high compatibility and efficiency across both Nvidia and AMD hardware, minor discrepancies may still occur due to platform-specific optimizations and compiler enhancements in CUDA and HIP, which are more native to their respective platforms.

3. Implementation Details

In this chapter, we will discuss the details of implementing the algorithms used for sparse matrix-vector multiplication (SpMV), including both the naive and merge-based approaches. Additionally, we will describe the implementation of algorithms for solving singular value decomposition (SVD) using a sparse matrix as input, specifically focusing on Jacobi rotations and Lanczos methods. Furthermore, we will detail the design and implementation of a 50/50 workload scheduler, which statically divides computational tasks between the CPU and integrated GPU to leverage both processing units in a heterogeneous system.

Our implementations will use *SYCL* and target heterogeneous systems composed of CPU + iGPU. The code is parameterized through native *SYCL* functions, which, when useful, will be mentioned. This means the code is written to work with most systems possible, making it the most portable.

As stated previously in an overview of the *SYCL* programming language in Section 2.6.3, *SYCL* uses **queue** objects as an abstraction connecting the host program and a single device. The runtime schedules and executes the actions asynchronously, keeping track of the action prerequisite in its scheduling, such as data availability. Therefore, all our implementations start by creating a queue where all the actions will be sent afterward.

```
1 sycl::queue q{sycl::default_selector_v};
```

Listing 3.1: Sycl Queue Creation

As a first parameter, we can pass a selector to specify what device the runtime should map the queue to, *SYCL* standard by default defines a few selectors based on a specific class of devices such as:

- default_selector: is the implementation-defined default device;
- cpu_selector: CPU device;
- gpu_selector: GPU device;
- accelerator_selector: an accelerator device, including FPGAs.

Another important instruction in *SYCL* is **wait**, which forces the queue to synchronize, ensuring all enqueued tasks are completed before proceeding. This can be useful when

the host needs to ensure all device computations are finished. For example (see listing 3.2):

```
q.wait();
```

Listing 3.2: Sycl Queue Synchronization

The **parallel_for** instruction defines parallel execution over various elements. It is an essential part of *SYCL*'s parallelism model, allowing tasks to be executed concurrently on a device. An example of usage is shown below (see listing 3.3):

```
q.submit([&](sycl::handler &h) {
    h.parallel_for(sycl::range<1>(n), [=](sycl::id<1> i) {
        // kernel code here
    });
});
```

Listing 3.3: Sycl parallel_for Example

In this example, the **parallel_for** takes a range that defines how many elements will be processed in parallel, and the lambda function specifies the operations to be performed on each element.

3.1. Compressed Sparse Row

All our implementations use the Compressed Sparse Row (CSR) structure to represent a sparse matrix.

As *SYCL* is an extension of CPP, which contains all the features of C, a simple **C struct** was used to represent this structure.

As stated previously in Section 2.1.2, the CSR is nothing more than three arrays representing the nonzero values of the matrix; as such, the code in C/CPP to represent it is the following:

```
typedef struct {
  int *row_offsets;
  int *column_indices;
  double *values;
} CompressedSparseRow;
```

Listing 3.4: Implementation of the data structure that supports CSR

To allocate memory in all of our implementations, we use the new *SYCL* Standard 2020 functionality, the *Unified Shared Memory (USM)* [31], that deals directly with point-based memory management. Alternatively, *SYCL* also has a buffer-accessor approach. A USM allocation using *SYCL* built-in function **shared_malloc** makes it accessible from both the host and the device, making it possible for the data to migrate freely without the need for the programmer intervention and synchronization between the host and the device.

```
1 CompressedSparseRow matrix;
2
3 // Memory Allocation
4 matrix->row_offsets = sycl::malloc_shared<int>(n + 1, q);
5 matrix->column_indices = sycl::malloc_shared<int>(nonzero, q);
6 matrix->values = sycl::malloc_shared<double>(nonzero, q);
```

Listing 3.5: Memory allocation for the data structures that support CSR

3.2. SpMV - A Naive Approach

As demonstrated in Algorithm 2, a sequential SpMV can be easily implemented using CSR format to represent a sparse matrix. For each output element (line 1), the algorithm loops through all entries in the corresponding row of the matrix (line 3). The *column_indices* retrieves the input vector's necessary entry, aggregating the result. This straightforward algorithm loads sequential entries from the *values* array and *column_indices* array, leading to an anticipated good cache behavior for the algorithm's core (line 4). Correspondingly, consecutive values in the *row_offsets* array are necessary. If the entries within the same row are close by, even the data loads from *x* will exhibit good cache behavior.

Algorithm 2 SpMV $y = A \cdot x$

1: **for** $i \leftarrow 0$ **to** matrix.row_offsets$[n]$ **do**
2: $\quad y_i \leftarrow 0$
3: $\quad$ **for** $k \leftarrow$ matrix.row_offsets$[i]$ **to** matrix.row_offsets$[i+1]$ **do**
4: $\quad\quad y_i \leftarrow y_i +$ matrix.values$[k] \times x[$matrix.column_indices$[k]]$
5: $\quad$ **end for**
6: $\quad y[i] \leftarrow y_i$
7: **end for**

It is pertinent to note that implementing a naive SpMV algorithm on the GPU resembles the approach used in the serial CPU algorithm. Starting from the serial algorithm, the process involves parallelizing the loop over the matrix rows (line 3 in algorithm 2).

The Kernel implementation in *SYCL* looks like this:

```
void spmv_row_parallel(queue &q, const CompressedSparseRow &matrix, const
    double *x, double *y) {
    const size_t num_rows = n;

    q.submit([&](handler &cgh) {
        cgh.parallel_for(range<1>(num_rows), [=](id<1> row_id) {
            double sum = 0.0;
            int start = matrix.row_offsets[row_id];
            int end = matrix.row_offsets[row_id + 1];

            for (int j = start; j < end; ++j) {
                sum += matrix.values[j] * x[matrix.column_indices[j]];
            }

            y[row_id] = sum;
        });
    }).wait();
}
```

Listing 3.6: Parallel SpMV Kernel Implementation

3.3. SpMV Merge

To implement the Merge Based SpMV, also called *CsrMV*, we used the already written code in CUDA available in the original paper as the base for our work; the algorithm is described in detail in section 2.2.1.

The implementation utilized involves three main steps. Initially, the elements of the output vector are initialized to zero. Subsequently, the sparse matrix and vector are multiplied. Finally, the rows of the output vector that span across multiple threads are adjusted.

The first two steps operate in parallel, using all available processors (threads). However, the final step involves performing a reduction. However, it could also be executed in parallel [10].

To initialize the output vector to zero, as shown in listing 3.7, we divide the array into equal parts, using *SYCL* **device** class abstraction. We query information using the **get_info** member function, specifically using the **max_compute_units** and the **max_work_group_size**, which allows us to compute the **thread_count** using the formula:

$$\text{thread\_count} = \text{compute\_units} \times \text{work\_group\_size}$$

It is important to note that the **thread_count** is always smaller than or equal to n (the number of elements to be processed). If the **thread_count** exceeds n, the code may behave incorrectly, as some threads would not have corresponding work. Therefore, in our

```
q.parallel_for(
  nd_range < 1 > (compute_units * work_group_size, work_group_size),
  [ = ](nd_item < 1 > item) {
    auto global_id = item.get_global_id(0);
    auto items_per_thread = (n + thread_count - 1) / thread_count;
    auto start = global_id * items_per_thread;
    auto stop = start + items_per_thread;

    for (auto i = start;
      (i < stop) && (i < n); i++) {
      y[i] = 0;
    }
  });

q.wait();
```

Listing 3.7: Parallel Output Vector Initialization

implementation and tests, we ensure that n is aligned with the **thread_count**. This alignment is crucial to avoid uneven workload distribution, where some threads might perform significantly more work than others. By aligning n and **thread_count**, we guarantee that the workload is properly balanced across all threads.

For the second step, the multiplication (shown in Listing 3.8), we start by identifying the merger lists and merge-path lengths for each thread (lines 4-7), a non-overlapping share of the overall work. We then identify the starting and ending diagonals (lines 9-18) and search for the corresponding 2D starting and ending coordinates within the merge grid. From lines 22-37, each thread executes the sequential SpMV Algorithm along its merge path. Finally, we save the running total and row ID for subsequent fix-up.

Finally, the last step is the fix-up (see listing 3.9), where we update the output vector values that spawned across multiple threads.

In this step, we must propagate the carry-over values from the previous parallel matrix-vector multiplication kernel. Each thread processes a portion of the sparse matrix, and since multiple threads may work on the same row, some partial dot products (i.e., partial sums) might remain unfinished. These partial sums must be added to the final result vector y to ensure correctness.

This fix-up phase is performed sequentially after the parallel computations have been completed. Each thread's carry-over values are stored in the **carry_value** array, and the corresponding row index is stored in the **carry_row** array. In the loop, we go through the carry-over values and add them to the appropriate row in y, ensuring no partial result is left unaccounted for. By performing this step sequentially, we avoid race conditions

```
q.parallel_for(
  nd_range < 1 > (compute_units * work_group_size, work_group_size),
  [ = ](nd_item < 1 > item) {
    auto global_id = item.get_global_id(0);

    int path_length = n + nonzero;
    int items_per_thread = (path_length + thread_count - 1) / thread_count;

    // Find starting and ending MergePath coordinates
    int diagonal = ((items_per_thread * tid) < path_length) ?
      (items_per_thread * tid) :
      path_length;
    int diagonal_end = ((diagonal + items_per_thread) < path_length) ?
      (diagonal + items_per_thread) :
      path_length;

    // Find the merge path range for the current thread (split point)
    MergeCoordinate path = MergePathBinarySearch(diagonal, matrix.row_offsets);
    MergeCoordinate path_end =
      MergePathBinarySearch(diagonal_end, matrix.row_offsets);

    double dot_product = 0.0;

    for (int i = 0; i < items_per_thread; i++) {
      // Move down (accumulate
      if (path.val_index < matrix.row_offsets[path.row_index + 1]) {
        dot_product += matrix.values[path.val_index] *
          x[matrix.column_indices[path.val_index]];
        path.val_index++;

      } else {
        // Move right (output row-total and reset)
        y[path.row_index] = dot_product;
        dot_product = 0;
        path.row_index++;
      }
    }

    carry_row[tid] = path_end.row_index;
    carry_value[tid] = dot_product;
  });

q.wait();
```

Listing 3.8: Parallel SpMV Merge Kernel Implementation

```
  for (int thread_id = 0; thread_id < thread_count - 1; thread_id++) {
    if (carry_row[thread_id] < n) {
      y[carry_row[thread_id]] += carry_value[thread_id];
    }
  }
```

Listing 3.9: Sequential Fix-Up Phase to Propagate Carry-Over Values from Parallel Computation

arising from multiple threads attempting to update the same row simultaneously during the parallel phase.

3.4. Jacobi Rotations

For this algorithm, we used an implementation by a colleague from the Faculty of Engineering of the University of Porto, Guilherme Sequeira, who used an OpenMP-based implementation as the foundation. The OpenMP implementation is available at `https://github.com/lixueclaire/Parallel-SVD`.

However, substantial work was done to rewrite and reorganize the code. Specifically, the code was refactored to use Unified Shared Memory (USM) for memory management instead of accessors, standardizing this approach for the entire implementation. Additionally, the code was refactored into separate headers to improve modularity, readability, and maintainability.

As input, a randomly generated matrix of size $N \times N$ is given, and the output matrices are U, V, and Σ, respectively.

The process iteratively applies rotations to zero out off-diagonal elements of the input matrix and then computes the singular values by normalizing the columns.

To zero out the off-diagonal elements (listing 3.10, lines 7-21), we use the Jacobi Rotations described in detail in section 2.3.1. Using an iterative process, we apply rotations to zero out the off-diagonal elements of the input matrix, which, in our case, is called U (a copy of the input matrix A); this continues until the convergence state (line 1) where all the off-diagonal elements of U are sufficiently small. At this point, the algorithm considers the matrix effectively diagonalized. As the epsilon value, we used the built-in `std::numeric_limits<T>::epsilon`.

Once the matrix U is sufficiently diagonalized (listings 3.11), the next step is to compute the singular values and finalize the transformation of U. For each column of U, the 2-*norm* (the square root of the sum of the squares of the elements) is computed. This norm represents the singular value associated with that column. The column of U is then normalized by dividing each component by this norm. The computed norms are stored in the vector S, which contains the singular values of the original matrix A. This algorithm performs a reduction operation to compute the sum of squares of the elements in a column of the matrix U (a critical step in normalization). Then, it uses the calculated value to normalize the matrix.

```
while (converge > epsilon) {
  for (size_t i = 1; i < m; i++) {
    for (size_t j = 0; j < i; j++) {
      RotationParams rp = get_rotation_params_parallel(q, U, m, n, i, j,
    converge);

      //Apply rotations on U and V
      q.submit([ & ](sycl::handler & h) {
        h.parallel_for(sycl::range < 1 > {
          n
        }, [ = ](sycl::id < 1 > idx) {
          double tan_val = U[idx * n + i];

          U[idx * n + i] = rp.cos_val * tan_val - rp.sin_val * U[idx * n + j];
          U[idx * n + j] = rp.sin_val * tan_val + rp.cos_val * U[idx * n + j];

          tan_val = V[idx * n + i];

          V[idx * n + i] = rp.cos_val * tan_val - rp.sin_val * V[idx * n + j];
          V[idx * n + j] = rp.sin_val * tan_val + rp.cos_val * V[idx * n + j];
        });
      }).wait();
    }
  }
}
```

Listing 3.10: Parallel Jacobi Rotations Kernel Implementation

```
for (int i = 0; i < m; i++) {
  double tan_val = 0;
  sycl::buffer < double > tan_buf {
    & tan_val, 1 };

  q.submit([ & ](sycl::handler & h) {
    auto tan_reduction = reduction(tan_buf, h, sycl::plus < > ());

    h.parallel_for(sycl::range < 1 > {
      n
    }, tan_reduction, [ = ](sycl::id < 1 > idx, auto & tan_accum) {
      tan_accum += pow(U[idx * n + i], 2);
    });
    tan_buf.get_host_access()[0];
  }).wait();

  tan_buf.get_host_access()[0];
  double get_tan_val = tan_buf.get_host_access()[0];
  tan_buf.get_host_access()[0] = sqrt(get_tan_val);

  for (int j = 0; j < n; j++) {
     auto num = U[j * n + i] / tan_val;

     U[j * n + i] = num;

     if (i == j) {
       S[i] = tan_val;
     }
  }
}
```

Listing 3.11: Parallel Singular Values Kernel Implementation

3.5. Lanczos

As stated previously, Lanczos is an algorithm that can calculate the SVD. Still, also, if we only implement the **for loop** without computing the eigenvalues and eigenvectors (line 10 of the pseudocode 1), we get a spectral decomposition where we get a tridiagonal Matrix T.

We hoped that performing this spectral decomposition before running the Jacobi Methods SVD described in the section 3.4 above would improve the previous algorithm's runtime.

To implement the spectral decomposition, we manually converted an already implemented CUDA code that is openly available on the web (`https://github.com/linhr/15618`) to the *SYCL* programming language.

A *CPP* class was implemented to represent the **tridiagonal matrix**, as shown in listing 3.12. The class design was inspired by the already implemented one in the original *CUDA* code. Specifically, the class contains two vectors: **alpha_**, which stores the diagonal elements, and **beta_**, which stores the subdiagonal elements.

The decision to use a class over a struct is based on the fact that the class encapsulates data and methods to access and modify these elements. The class provides methods to resize the vectors and retrieve or modify individual elements. By encapsulating these operations in a class, we ensure that the internal structure of the matrix (i.e., the size of the diagonal and subdiagonal vectors) is consistent and that any resizing or element access is done through controlled interfaces.

The class design includes the following public methods:

- `size()`: Returns the size of the matrix (i.e., the number of diagonal elements, n).
- `resize(int n)`: Resizes both the diagonal and subdiagonal vectors to accommodate changes in matrix dimensions.
- Getter and setter methods for accessing and modifying individual diagonal (`alpha(int i)`) and subdiagonal (`beta(int i)`) elements.
- `alpha_data()` and `beta_data()`: Provide direct access to the raw data pointers of the diagonal and subdiagonal vectors.

The Lanczos spectral decomposition (listing 3.13) starts with allocating the required memory for the vectors and arrays. This includes the previous vector `x_prev`, the current vector

```
1  class tridiagonal_matrix {
2  public:
3      explicit tridiagonal_matrix(int n)
4              : alpha_(n), beta_(n - 1) {}
5
6      [[nodiscard]] int size() const { return static_cast<int>(alpha_.size()); }
7
8      void resize(int n) {
9          alpha_.resize(n);
10         beta_.resize(n - 1);
11     }
12
13     [[nodiscard]] const double &alpha(int i) const { return alpha_[i]; }
14
15     [[nodiscard]] const double &beta(int i) const { return beta_[i]; }
16
17     double &alpha(int i) { return alpha_[i]; }
18
19     double &beta(int i) { return beta_[i]; }
20
21     [[nodiscard]] const double *alpha_data() const { return alpha_.data(); }
22
23     [[nodiscard]] const double *beta_data() const { return beta_.data(); }
24
25     double *alpha_data() { return alpha_.data(); }
26
27     double *beta_data() { return beta_.data(); }
28
29 private:
30     std::vector<double> alpha_;
31     std::vector<double> beta_;
32 };
```

Listing 3.12: Tridiagonal Matrix class

`y`, and a scratch array for intermediate results (lines 16-18). Additionally, the row pointers, column indices, and non-zero values of the sparse matrix are copied to device memory, ensuring the CSR matrix data is available for computations on the device. The initial vector `x` is also copied to the device from the input vector `v` (line 31).

The core of the Lanczos iteration is performed inside a loop that runs for the specified number of steps (line 41). In each step, the matrix-vector multiplication is carried out first. The current Lanczos vector `x` is multiplied by the sparse matrix `M`, producing a new vector `y`. This operation is handled by the `warp_multiply_kernel`, which performs the multiplication in parallel, taking advantage of *SYCL*'s parallelism (line 43).

Next, the diagonal element `alpha_i` of the tridiagonal matrix is computed by taking the dot product of the vectors `y` and `x` (line 46-47). This value, which is stored in `result.alpha(i)`, represents the projection of `y` onto `x` and forms part of the tridiagonal matrix.

After this, the vector `y` is updated to ensure orthogonality concerning the previous Lanczos

vectors. This update involves subtracting `alpha_i * x_i` from `y`, and if this is not the first iteration, subtracting `beta_(i-1) * x_(i-1)` as well (lines 50-53). This process is crucial to maintaining the orthogonality condition required by the Lanczos algorithm.

The subdiagonal element `beta_(i+1)` of the tridiagonal matrix is then computed by calculating the norm of the updated vector `y` (line 43). This norm is obtained by taking the square root of the dot product of `y` with itself. The value of `beta_(i+1)` is stored in `result.beta(i)`, representing the connection between consecutive Lanczos vectors in the tridiagonal matrix.

Finally, the new Lanczos vector `x_(i+1)` is generated by normalizing `y` with `beta_(i+1)` (line 62). This normalization ensures that the new Lanczos vector has a unit length, which the Lanczos process requires. The vectors `x` and `y` are then swapped, preparing for the next iteration of the loop.

After completing all iterations, the tridiagonal matrix is resized to match the number of steps performed (line 72). The device memory allocated for the vectors and matrix data, including `x_prev`, `y`, `scratch`, `row_ptr`, and `col_ind`, is freed (lines 73-77), ensuring efficient memory management. The final tridiagonal matrix, which contains the computed `alpha` and `beta` values, is then returned as the result of the function (line 79).

```
1  // Function to perform Lanczos iterations using SYCL
2  tridiagonal_matrix lanczos_iteration(sycl::queue & q,
3    const csr_matrix & m,          // Input sparse matrix in CSR format
4    const std::vector < double > & v, // Initial vector for Lanczos iteration
5    const int steps) {             // Number of Lanczos steps to perform
6
7    // Initialize a tridiagonal matrix to store the results of the Lanczos
       iterations
8    tridiagonal_matrix result(steps + 1);
9
10   int rows = m.row_size();        // Number of rows in the matrix
11   int cols = m.col_size();        // Number of columns in the matrix
12   int nonzeros = m.nonzeros();    // Number of non-zero elements in the matrix
13   const int blocks = (rows + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK; //
       Number of blocks needed
14
15   // Allocate device memory
16   auto * x_prev = sycl::malloc_device < double > (cols, q); // Memory for x_(i
       -1)
17   auto * y = sycl::malloc_device < double > (cols, q);     // Memory for y_i
```

```
18  auto * scratch = sycl::malloc_device < double > (blocks, q); // Scratch space
      for intermediate results
19
20  // Allocate and initialize device memory for matrix data
21  int * row_ptr = sycl::malloc_device < int > (rows + 1, q); // Row pointers
      for CSR matrix
22  q.memcpy(row_ptr, m.row_ptr_data(), sizeof(int) * (rows + 1)); // Copy row
      pointer data
23
24  int * col_ind = sycl::malloc_device < int > (nonzeros, q); // Column indices
      for CSR matrix
25  q.memcpy(col_ind, m.col_ind_data(), sizeof(int) * nonzeros); // Copy column
      index data
26
27  auto * values = sycl::malloc_device < double > (nonzeros, q); // Non-zero
      values for CSR matrix
28  q.memcpy(values, m.values_data(), sizeof(double) * nonzeros); // Copy values
      data
29
30  auto * x = sycl::malloc_device < double > (cols, q); // Memory for x_i
31  q.memcpy(x, v.data(), sizeof(double) * cols); // Copy initial vector to
      device memory
32
33  // Determine the optimal group size for kernel execution
34  const int row_nonzeros = nonzeros / rows;
35  int group_size = row_nonzeros > 16 ? 32 : 16;
36  group_size = row_nonzeros > 8 ? group_size : 8;
37  group_size = row_nonzeros > 4 ? group_size : 4;
38  group_size = row_nonzeros > 2 ? group_size : 2;
39
40  // Execute Lanczos iterations
41  for (int i = 0; i < steps; i++) {
42    // Compute y_i = M * x_i using warp-based matrix-vector multiplication
43    warp_multiply_kernel(q, group_size, rows, row_ptr, col_ind, values, x, y);
44
45    // Compute alpha_i = y_i^T * x_i (dot product of y_i and x_i)
46    double product = device_dot_product(q, rows, x, y, scratch);
47    result.alpha(i) = product; // Store alpha_i
48
49    // Update y_i = y_i - alpha_i * x_i - beta_i * x_(i-1)
50    saxpy_inplace_kernel(q, rows, y, x, -product);
51    if (i > 0) {
```

```
52        saxpy_inplace_kernel(q, rows, y, x_prev, -result.beta(i - 1));
53      }
54
55      // Swap pointers for the next iteration
56      std::swap(x, x_prev);
57
58      // Compute beta_(i+1) = ||y_i|| (norm of y_i)
59      result.beta(i) = std::sqrt(device_dot_product(q, rows, y, y, scratch));
60
61      // Normalize y_i to get x_(i+1) = y_i / beta_(i+1)
62      multiply_inplace_kernel(q, rows, y, 1 / result.beta(i));
63
64      // Swap pointers for the next iteration
65      std::swap(x, y);
66    }
67
68    // Wait for all operations to complete
69    q.wait();
70
71    // Resize the result to the number of steps and free allocated memory
72    result.resize(steps);
73    sycl::free(x_prev, q);
74    sycl::free(y, q);
75    sycl::free(scratch, q);
76    sycl::free(row_ptr, q);
77    sycl::free(col_ind, q);
78
79    return result; // Return the tridiagonal matrix containing the results
80  }
```

Listing 3.13: Parallel Lanczos Spectral Decomposition Kernel Implementation

```
1  // SAXPY operation (Single-Precision A·X Plus Y) kernel
2  void saxpy_inplace_kernel(sycl::queue &q, const int n, double *y, const double
       *x, const double a) {
3
4      // Calculate the total number of threads needed, rounded up to the nearest
       multiple of THREADS_PER_BLOCK
5      const int threads = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK *
       THREADS_PER_BLOCK;
6      sycl::range<1> gws (threads); // Global Work Size (total number of threads)
7      sycl::range<1> lws (THREADS_PER_BLOCK); // Local Work Size (number of
       threads per work-group)
```

```
8
9       // Submit command group to the queue
10      q.submit([&] (sycl::handler &cgh) {
11          cgh.parallel_for(
12                  sycl::nd_range<1>(gws, lws), [=] (sycl::nd_item<1> item) {
13                      int index = static_cast<int>(item.get_global_id(0)); //
        Global ID of the thread
14                      if (index < n) y[index] += a * x[index]; // SAXPY operation
15                  });
16      });
17  }
18
19  // In-place multiplication kernel
20  void multiply_inplace_kernel(sycl::queue &q, const int n, double *x, const
        double k) {
21
22      // Calculate the total number of threads needed, rounded up to the nearest
        multiple of THREADS_PER_BLOCK
23      const int threads = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK *
        THREADS_PER_BLOCK;
24      sycl::range<1> gws (threads); // Global Work Size (total number of threads)
25      sycl::range<1> lws (THREADS_PER_BLOCK); // Local Work Size (number of
        threads per work-group)
26
27      // Submit command group to the queue
28      q.submit([&] (sycl::handler &cgh) {
29          cgh.parallel_for(
30                  sycl::nd_range<1>(gws, lws), [=] (sycl::nd_item<1> item) {
31                      int index = static_cast<int>(item.get_global_id(0)); //
        Global ID of the thread
32                      if (index < n) x[index] *= k; // In-place multiplication
33                  });
34      });
35  }
36
37  // Warp-based matrix-vector multiplication kernel
38  void warp_multiply_kernel(sycl::queue &q, const int group_size, const int rows,
         const int *row_ptr, const int *col_ind,
39                            const double *values, const double *x, double *y) {
40
41      const int groups_per_block = THREADS_PER_BLOCK / group_size; // Number of
        groups per block
```

```
42      const int multiply_blocks = (rows + groups_per_block - 1) /
        groups_per_block; // Number of blocks needed
43      sycl::range<1> gws (multiply_blocks * THREADS_PER_BLOCK); // Global Work
        Size (total number of threads)
44      sycl::range<1> lws (THREADS_PER_BLOCK); // Local Work Size (number of
        threads per work-group)
45
46      // Submit command group to the queue
47      q.submit([&] (sycl::handler &cgh) {
48          sycl::local_accessor<double, 1> result(sycl::range<1>(THREADS_PER_BLOCK
        ), cgh); // Local memory for each work-group
49          cgh.parallel_for(
50                  sycl::nd_range<1>(gws, lws), [=] (sycl::nd_item<1> item) {
51                      int index = static_cast<int>(item.get_global_id(0)); //
        Global ID of the thread
52                      int lid = static_cast<int>(item.get_local_id(0)); // Local
        ID within the work-group
53                      int r = index / group_size; // Row index
54                      int lane = index % group_size; // Lane index within the
        warp
55
56                      result[lid] = 0; // Initialize result
57                      if (r < rows) {
58                          int start = row_ptr[r];
59                          int end = row_ptr[r + 1];
60                          for (int i = start + lane; i < end; i += group_size)
61                              result[lid] += values[i] * x[col_ind[i]]; //
        Compute partial results
62
63                          // Warp-wide reduction (summing up results)
64                          int half = group_size / 2;
65                          while (half > 0) {
66                              if (lane < half) result[lid] += result[lid + half];
67                              half /= 2;
68                          }
69                          if (lane == 0) y[r] = result[lid]; // Write the final
        result
70                      }
71                  });
72      });
73  }
74
```

```
75 // Dot product kernel
76 double device_dot_product(sycl::queue &q, const int n, const double *x, const
       double *y, double *z) {
77
78     // Calculate number of blocks needed
79     const int blocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
80     const int threads = blocks * THREADS_PER_BLOCK; // Total number of threads
81     sycl::range<1> gws (threads); // Global Work Size (total number of threads)
82     sycl::range<1> lws (THREADS_PER_BLOCK); // Local Work Size (number of
       threads per work-group)
83
84     // Submit command group to the queue
85     q.submit([&] (sycl::handler &cgh) {
86         sycl::local_accessor<double, 1> result (sycl::range<1>(
       THREADS_PER_BLOCK), cgh); // Local memory for each work-group
87         cgh.parallel_for(
88                 sycl::nd_range<1>(gws, lws), [=] (sycl::nd_item<1> item) {
89                     int index = static_cast<int>(item.get_global_id(0)); //
       Global ID of the thread
90                     int lid = static_cast<int>(item.get_local_id(0)); // Local
       ID within the work-group
91                     int bid = static_cast<int>(item.get_group(0)); // Block ID
92
93                     if (index < n)
94                         result[lid] = x[index] * y[index]; // Compute partial
       dot product
95                     else
96                         result[lid] = 0;
97
98                     item.barrier(sycl::access::fence_space::local_space); //
       Synchronize threads within the work-group
99
100                    // Perform reduction to sum up results within the work-
       group
101                    int half = THREADS_PER_BLOCK / 2;
102                    while (half > 0) {
103                        if (lid < half)
104                            result[lid] += result[lid + half];
105                        item.barrier(sycl::access::fence_space::local_space);
106                        half /= 2;
107                    }
```

```
108                     if (lid == 0) z[bid] = result[0]; // Write result from the
    first thread in each work-group
109                 });
110     });
111
112     // Transfer result from device to host
113     std::vector<double> host_scratch(blocks);
114
115     q.memcpy(host_scratch.data(), z, sizeof(double) * blocks); // Copy result
    to host
116
117     double result(0);
118
119     q.wait(); // Wait for all operations to complete
120
121     for (int i = 0; i < blocks; i++) {
122         result += host_scratch[i]; // Sum up all partial results
123     }
124     return result;
125 }
```

Listing 3.14: Kernels for Linear Algebra Operations. This listing includes implementations for several fundamental linear algebra operations using SYCL. Specifically, it contains kernels for (1) SAXPY (Single-Precision A·X Plus Y), (2) in-place vector multiplication, (3) warp-based matrix-vector multiplication, and (4) dot product computation.

3.6. 50/50 Scheduler for Jacobi Rotations

For the 50/50 scheduler, we extended the single-device Jacobi kernel to use both the CPU and iGPU for parallel computation. This scheduler allocates half of the workload to each processing unit, aiming to divide matrix computations between the CPU and iGPU, with the goal of achieving a balanced workload across both devices and increase performance.

The adaptation required refactoring the Jacobi rotation implementation to incorporate both CPU and iGPU task submission queues, enabling parallel execution. Similar to the single-kernel implementation, the 50/50 scheduler uses the *SYCL* feature to obscure the manage memory automatically across both devices, as needed, facilitating shared data access without requiring explicit data transfers.

The workflow of the 50/50 scheduler begins by partitioning the matrix into two sections, each assigned to a separate processing queue. Specifically, the CPU handles the first half of each row in the matrix, while the iGPU processes the second half. This division is static, meaning that each device is allocated an equal portion of data without dynamically adjusting based on runtime performance feedback.

The algorithm iteratively applies rotations to zero out off-diagonal elements in a manner similar to the single-kernel implementation. However, in this case, the 50/50 scheduler applies these rotations in parallel across the two processing units. Within each iteration, the CPU and iGPU each execute a portion of the rotation transformations on the matrix U, a copy of the input matrix A, until the convergence condition is met. The convergence state is determined when all off-diagonal elements of U fall below a specified threshold, set by the epsilon value `std::numeric_limits<T>::epsilon`, as previously.

Due to the need for synchronization between the two devices, explicit `wait()` calls are inserted at the end of each iteration cycle. This ensures that the CPU and iGPU complete their respective portions of the computation before proceeding to the next iteration.

After the matrix U is sufficiently diagonalized, the next step is to compute the singular values and finalize the transformation of U. Similar to the single-device kernel, this involves calculating the *2-norm* of each column, which is done through a reduction operation. This operation is performed on the iGPU, which is well-suited for parallel reductions, to accumulate the square values across the elements of each column. The resulting norms are stored in the vector S, which contains the singular values of the original matrix A.

```
while (converge > epsilon) {
  for (size_t i = 1; i < m; i++) {
    for (size_t j = 0; j < i; j++) {
      RotationParams rp = get_rotation_params_parallel(cpu_queue, U, m, n, i, j
    , converge);

      size_t half_n = n / 2;

      // Apply rotations on U and V
      cpu_queue.submit([&](sycl::handler &h) {
        h.parallel_for(sycl::range<1>{half_n}, [=](sycl::id<1> idx) {
          double tan_val = U[idx * n + i];
          U[idx * n + i] = rp.cos_val * tan_val - rp.sin_val * U[idx * n + j];
          U[idx * n + j] = rp.sin_val * tan_val + rp.cos_val * U[idx * n + j];

          tan_val = V[idx * n + i];
          V[idx * n + i] = rp.cos_val * tan_val - rp.sin_val * V[idx * n + j];
          V[idx * n + j] = rp.sin_val * tan_val + rp.cos_val * V[idx * n + j];
        });
      });

      gpu_queue.submit([&](sycl::handler &h) {
        h.parallel_for(sycl::range<1>{n - half_n}, [=](sycl::id<1> idx) {
          double tan_val = U[(idx + half_n) * n + i];
          U[(idx + half_n) * n + i] = rp.cos_val * tan_val - rp.sin_val * U[(
    idx + half_n) * n + j];
          U[(idx + half_n) * n + j] = rp.sin_val * tan_val + rp.cos_val * U[(
    idx + half_n) * n + j];

          tan_val = V[(idx + half_n) * n + i];
          V[(idx + half_n) * n + i] = rp.cos_val * tan_val - rp.sin_val * V[(
    idx + half_n) * n + j];
          V[(idx + half_n) * n + j] = rp.sin_val * tan_val + rp.cos_val * V[(
    idx + half_n) * n + j];
        });
      });

      cpu_queue.wait();
      gpu_queue.wait();
    }
  }
}
```

Listing 3.15: Parallel Jacobi Rotations Kernel Implementation with 50/50 Scheduler

4. Experimental Results

This chapter presents the experimental results obtained from the work developed; the experiments were conducted on a Thinkpad T460s laptop with an Intel Core i5 processor, 12 GB of RAM, and an integrated Intel HD Graphics 520 GPU (corresponding to the Generation 9 studied and described in detail in Section 2.5). The *SYCL* compiler employed for the development and execution of our code was the latest Intel oneAPI DPC++ Compiler (version 2024.2.0). Our testing environment was established on a 64-bit Fedora Linux 40 operating system. All experimental setups were executed within Docker containers, allowing for an isolated and reproducible runtime environment, for the SVD implementations we used 10 iterations to benchmark it, while for the SpMV we used a 1000 iterations, this decision had to do with the runtime differences in each algorithm.

4.1. SpMV

The following runtimes were obtained from 1000 iterations, and the average of these execution times was subsequently calculated for each case. Additionally, our sparse matrices consistently contain 20% nonzero values to approximate real-world datasets closely.

In the **SpMV Merge based**, the integrated GPU consistently performs better than the sequential method, with the gap increasing as the matrix size increases. As we can state, as an example, for the matrix dimensions of 2000000×2000000, the integrated GPU completes the kernel in 0.00146 seconds compared to 0.00189 seconds for the sequential implementation, representing a 22.8% performance improvement. For larger matrix dimensions of 25600000×25600000, the improvement increases to 26.7%, with the integrated GPU taking 0.32012 seconds compared to 0.34662 seconds in the sequential implementation. These performance gains follow the trend of the usage of the integrated GPU becoming more effective as the problem size increases, as shown in Figure 4.2. Discussing the scaling behavior, the integrated GPU shows better scalability; while the sequential execution time increases by 230.4, the integrated GPU increase is only 208.4, as depicted in Table 4.1.

As for the **SpMV Row**, the integrated GPU advantage is even more significant. For example, at 200000×200000 matrix dimensions, the integrated GPU computes it in 0.00056 seconds compared to 0.00195 seconds for the sequential method, marking a 71.4% improvement. For the largest tested matrix size (25600000×25600000), the integrated GPU

shows a 74.1% improvement. Figure 4.1 illustrates this trend, demonstrating how the row-based method benefits from GPU acceleration. Sequential execution time increases by 230.7 as matrix sizes scale from 200000×200000 to 25600000×25600000. At the same time, the integrated GPU shows a more controlled increase factor (218.8), as highlighted in Table 4.2. Both implementations exhibit a roughly linear increase in execution time, but the integrated GPU implementation shows better scalability.

These observations are reinforced by the experimental results of the original SpMV-Based paper [10], showing that merge-based approaches have a much stronger correlation between runtime and problem size than the row-based method (0.79 compared to 0.31). This higher correlation indicates that the merge-based method scales more predictably as the matrix size increases, as shown in Figure 4.3, meaning that as the matrix size grows, the merge-based runtime increases more consistently, leading to stable performance scaling.

It also states that the merge-based method is less sensitive to variation in the structure of the matrix, showing a much lower correlation of **FLOPS** (floating point operations per second) to row-length variation (-0.02) when compared to the row-based method (-0.24).

The data shown in Tables 4.1 and 4.2 contradicts the literature. However, due to our system's memory limitations, we could not replicate the results for larger matrix sizes used in the original paper. Despite this, insights from the literature suggest that, despite being slower in some cases, the merge-based algorithm is more predictable and stable when dealing with larger problem sizes.

The merge-based SpMV implementation provides more consistent and predictable scaling behavior, making it a better choice for applications dealing with large, irregular matrices. Its lower sensitivity to matrix structure allows it to maintain high efficiency across various matrix sizes. Meanwhile, the row-based SpMV method is ideal for applications with smaller, more regular matrices, where its simplicity and higher performance for certain sizes can provide an advantage. However, as the matrix size increases, the limitations of the row-based method become more apparent due to its higher sensitivity to matrix structure, leading to performance degradation.

In conclusion, both SpMV methods offer distinct advantages depending on the problem. The merge-based method excels in scenarios involving large, irregular matrices due to its predictable scaling and reduced sensitivity to matrix structure variations. On the other hand, the row-based method can outperform in smaller or more structured matrices, but

its performance scales less efficiently as matrix sizes grow.

Tables 4.1 and 4.2 provide a comprehensive comparison of the two methods' performance across different matrix sizes, while Figures 4.2, 4.1, and 4.3 highlight the performance trends and scalability differences between the methods.

Matrix Dimensions	**Sequential (sec)**	**IGPU (sec)**
200000×200000	1.89317×10^{-3}	1.46217×10^{-3}
400000×400000	3.85071×10^{-3}	3.109×10^{-3}
800000×800000	8.51911×10^{-3}	7.33242×10^{-3}
1600000×1600000	1.83374×10^{-2}	1.60051×10^{-2}
3200000×3200000	3.89465×10^{-2}	3.50555×10^{-2}
6400000×6400000	8.4043×10^{-2}	7.29353×10^{-2}
12800000×12800000	1.86079×10^{-1}	1.59517×10^{-1}
25600000×25600000	4.3662×10^{-1}	3.20122×10^{-1}

Table 4.1: Execution Times for **SpMV Merge** Based Sequential and Integrated GPU

Matrix Dimensions	**Sequential (sec)**	**IGPU (sec)**
200000×200000	1.94691×10^{-3}	5.56883×10^{-4}
400000×400000	4.08462×10^{-3}	1.00978×10^{-3}
800000×800000	9.2999×10^{-3}	2.15088×10^{-3}
1600000×1600000	2.24058×10^{-2}	5.54045×10^{-3}
3200000×3200000	4.74906×10^{-2}	1.25957×10^{-2}
6400000×6400000	9.89569×10^{-2}	2.60905×10^{-2}
12800000×12800000	2.10247×10^{-1}	5.74857×10^{-2}
25600000×25600000	4.48547×10^{-1}	1.16034×10^{-1}

Table 4.2: Execution Times for **SpMV Row** Based Sequential and Integrated GPU

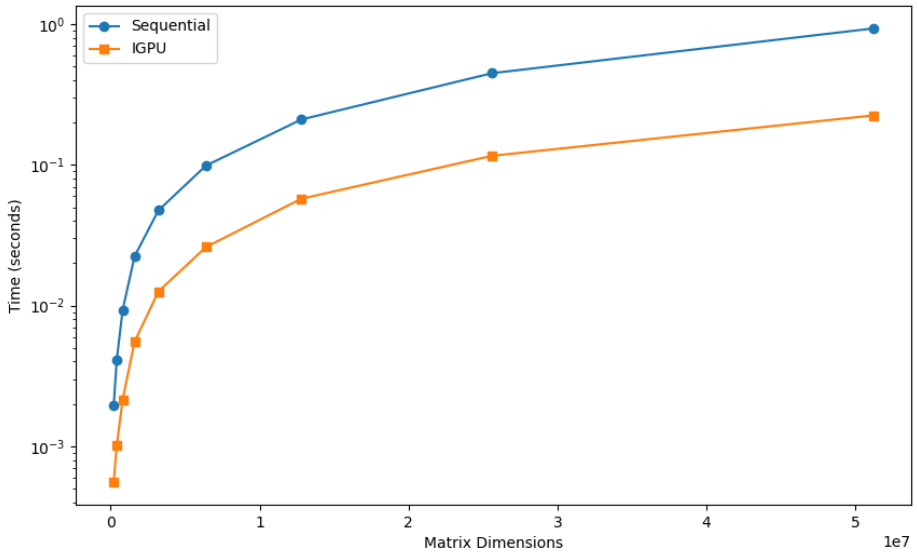

Figure 4.1: SpMV Row Results

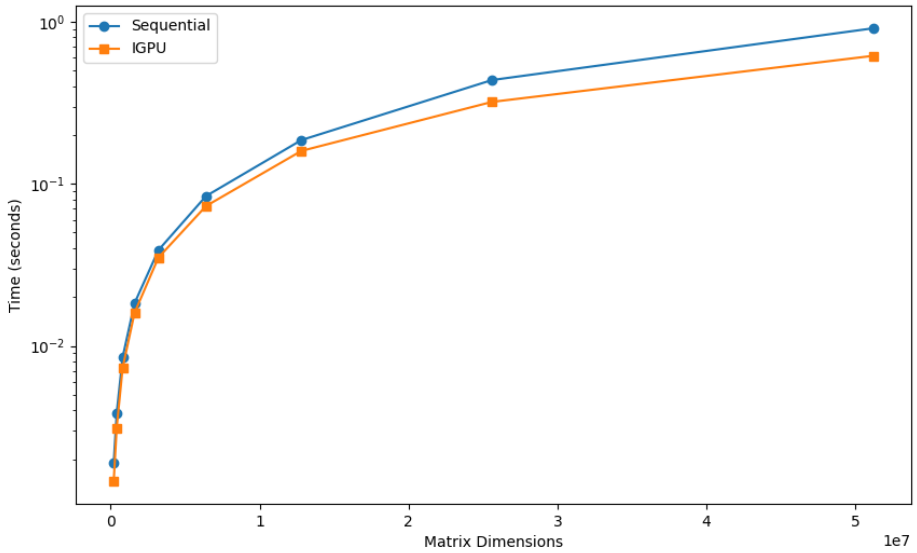

Figure 4.2: SpMV Merge Results

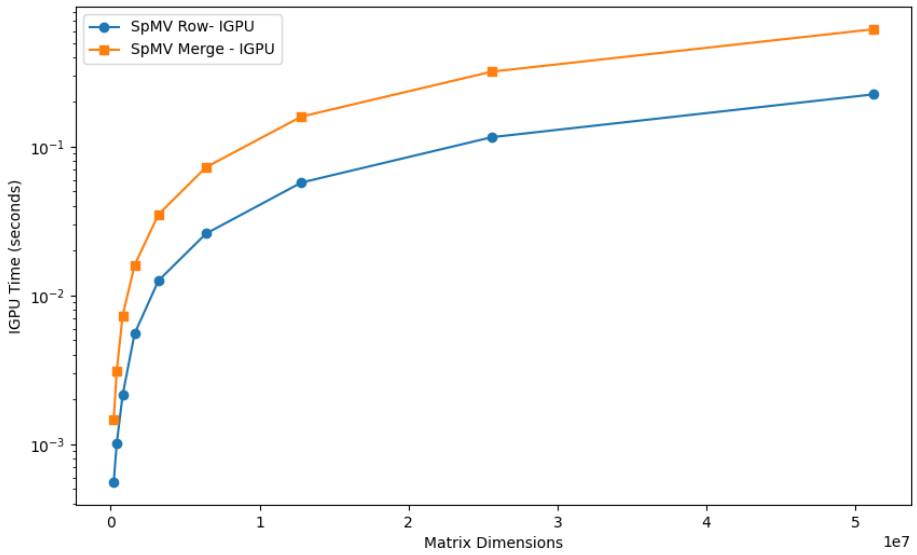

Figure 4.3: SpMV Runtime in the Integrated GPU

4.2. SVD

The following runtimes were obtained from 10 iterations, and the average execution times were subsequently calculated for each case. Additionally, our sparse matrices consistently contain 20% nonzero values to approximate real-world datasets closely. All the matrices, either dense or sparse, were created at runtime randomly. All times are expressed in milliseconds.

The Jacobi rotations method applied to dense matrices shows the highest execution time among all implementations for both the iGPU and CPU, as illustrated in Table 4.3 and Figure 4.4. The runtimes for the iGPU show a significant exponential increase as the matrix size grows, starting from a runtime of 1.5469×10^4 milliseconds for a 128×128 matrix to about 1.0222461×10^7 milliseconds (approximately 170 minutes) for a 2048×2048 matrix.

Similarly, the CPU starts with a slower runtime progression compared to the iGPU for smaller matrices. For a 128×128 matrix, the CPU takes 2.1777×10^4 milliseconds, which is higher than the iGPU. However, as the matrix size increases, the CPU eventually surpasses the iGPU in performance. Starting from a 512×512 matrix, the CPU becomes more efficient, and for the largest matrix size of 2048×2048, the CPU runtime is 1.0772122×10^7 milliseconds (approximately 179.5 minutes), which is slightly slower than the iGPU.

This exponential increase in runtimes for both hardware platforms highlights the limitations of the Jacobi rotations method for large dense matrices. The algorithm's complexity grows significantly with the matrix size, making it impractical for large-scale problems. Despite the accurate results it provides, the pure Jacobi method does not scale well, making it infeasible for handling large datasets.

In contrast, the Lanczos with Jacobi rotations method, applied to sparse matrices, provides a much more efficient solution. As seen in Table 4.4 and Figure 4.5, this hybrid approach demonstrates significantly lower runtimes, particularly for larger matrices. For example, the iGPU takes approximately 1.945162×10^6 milliseconds (about 54 minutes) to compute the SVD for a 2048×2048 matrix, which is dramatically faster than the 1.0222461×10^7 milliseconds (170 minutes) required by the Jacobi rotations method for dense matrices of the same size. Similarly, the CPU completes the computation in 2.022309×10^6 milliseconds (around 56.2 minutes) for the hybrid Lanczos approach, much faster than the 1.0772122×10^7 milliseconds (179.5 minutes) it takes for the pure Jacobi method.

For smaller matrices, such as the 128×128 case, the iGPU completes the Lanczos method in 3.712×10^3 milliseconds, while the CPU takes 5.167×10^3 milliseconds. However, as matrix size increases, the CPU closes the gap and becomes increasingly competitive with the iGPU, particularly for larger matrices. This suggests that while the iGPU benefits from parallelism for smaller tasks, the CPU's scalability allows it to maintain efficiency for larger matrix sizes.

Comparing the two methods, the Lanczos with Jacobi rotations approach offers an advantage in terms of scalability and performance, especially for large sparse matrices. The Jacobi rotations method, while providing an exact and complete solution for the SVD of dense matrices, becomes inefficient as matrix size grows, with runtimes increasing exponentially. On the other hand, the Lanczos method reduces the complexity by first applying spectral decomposition, followed by Jacobi rotations, leading to faster execution times, particularly for large matrix sizes.

In terms of hardware performance, the iGPU shows better results for smaller matrices across both methods, but as the matrix size grows, the CPU demonstrates more efficient performance. This is particularly evident in the pure Jacobi rotations method, where the CPU surpasses the iGPU for matrix sizes of 512×512 and larger. In the hybrid Lanczos method, both the CPU and iGPU perform similarly for larger matrices, but the CPU

remains slightly slower than the iGPU.

In conclusion, both the Jacobi rotations method and the hybrid Lanczos with Jacobi rotations approach exhibit an exponential increase in runtime as matrix size increases. However, the Jacobi-based method shows a significantly worse rate of increase, making it unsuitable for large matrix sizes. The Lanczos hybrid method, by contrast, offers a more scalable solution, particularly for sparse matrices. The iGPU demonstrates strong performance for small to medium-sized matrices, while the CPU becomes more efficient for larger matrices, especially in the Jacobi rotations method. The Lanczos with Jacobi approach ultimately strikes a balance between accuracy and efficiency, making it a more practical solution for handling large matrix computations.

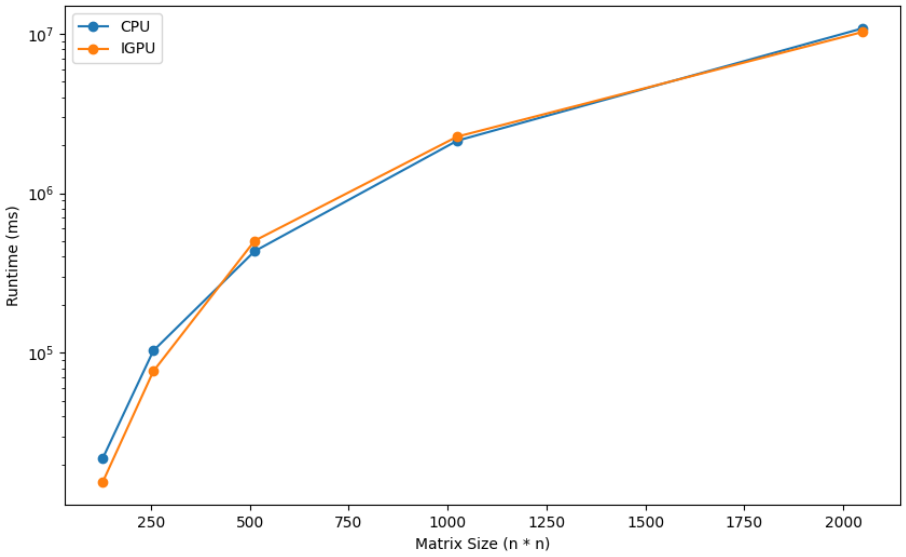

Figure 4.4: Jacobi Rotations SVD Execution Time

Matrix Size	**iGPU**	**CPU**
128×128	1.5469×10^4	2.1777×10^4
256×256	7.6558×10^4	1.02954×10^5
512×512	5.03292×10^5	4.32728×10^5
1024×1024	2.259873×10^6	2.131685×10^6
2048×2048	1.0222461×10^7	1.0772122×10^7

Table 4.3: Runtimes for Jacobi Rotation on Intel(R) HD Graphics 520 and Intel(R) Core(TM) i5-6300U CPU for Different Matrix Sizes

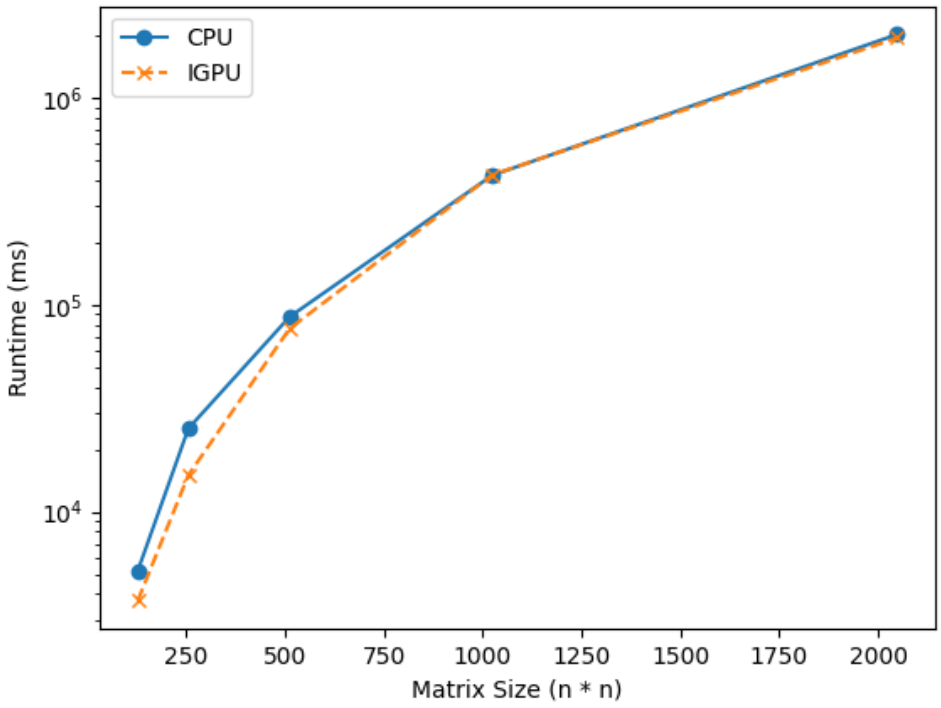

Figure 4.5: Lanczos as Spectral Decomposition followed by Jacobi Rotations SVD

Matrix Size	CPU	iGPU
128×128	5.167×10^3	3.712×10^3
256×256	2.5196×10^4	1.5006×10^4
512×512	8.7340×10^4	7.6673×10^4
1024×1024	4.21426×10^5	4.23298×10^5
2048×2048	2.022309×10^6	1.945162×10^6

Table 4.4: Execution Times for SVD Kernel using Lanczos Spectral Decomposition and Jacobi Rotations on Intel(R) Core(TM) i5-6300U CPU and Intel(R) HD Graphics 520

4.3. Simultaneous Execution of Independent SVD Kernels on CPU and Integrated GPU

In this subsection, we focus on the performance of simultaneous execution of SVD kernels on both the CPU and the iGPU. The goal is to assess how these two processing units handle identical workloads when running in parallel, and what implications this has for heterogeneous computing.

The results, presented in Table 4.5 and visualized in Figure 4.6, indicate that the integrated GPU consistently outperforms the CPU for smaller matrix sizes. For instance, at a matrix size of 128×128, the iGPU completes the task in 3.811×10^3 ms, while the CPU takes 5.084×10^3 ms. This represents a **25% performance improvement** when utilizing the

iGPU in simultaneous execution, as did in the previously sections. The advantage of the iGPU is further emphasized at matrix sizes of 256 and 512, where it continues to deliver faster runtimes, specifically 1.5050×10^4 ms and 7.4206×10^4 ms, respectively, compared to the CPU's 2.1714×10^4 ms and 1.00616×10^5 ms.

This increase in runtime, when compared to the execution times of kernels running one at a time (4.4 and Figure 4.7), is expected. During simultaneous execution, the CPU still needs to send instructions to the iGPU, which introduces additional overhead. This communication cost between the CPU and iGPU, particularly when managing data transfers and coordinating the execution of the same kernel on both devices, contributes to the increased overall runtime.

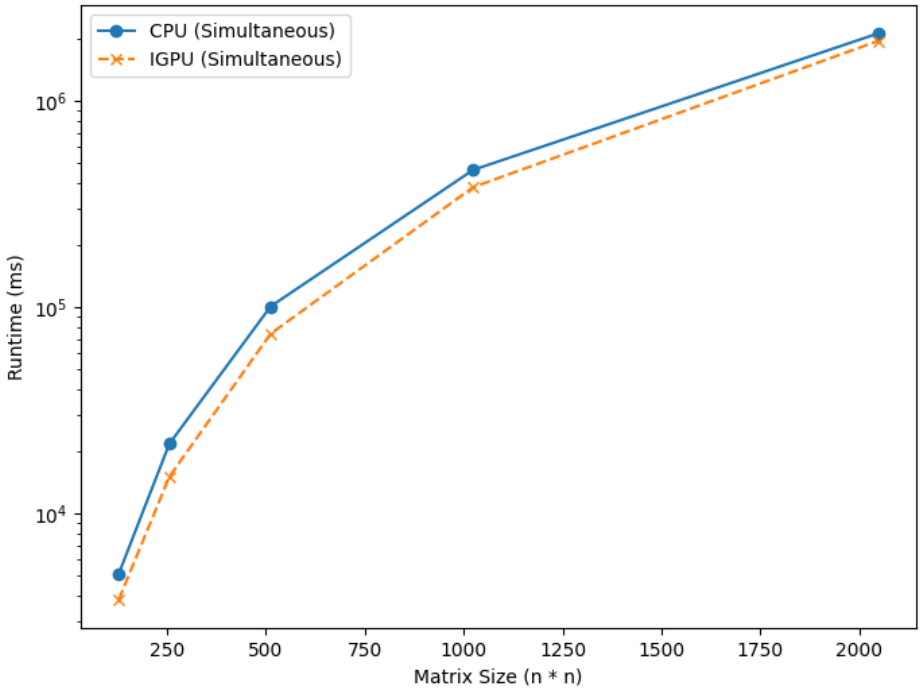

Figure 4.6: Comparison of Execution Times for Simultaneous Lanczos Hybrid SVD Kernel Execution on CPU and Integrated GPU across varying Matrix Sizes.

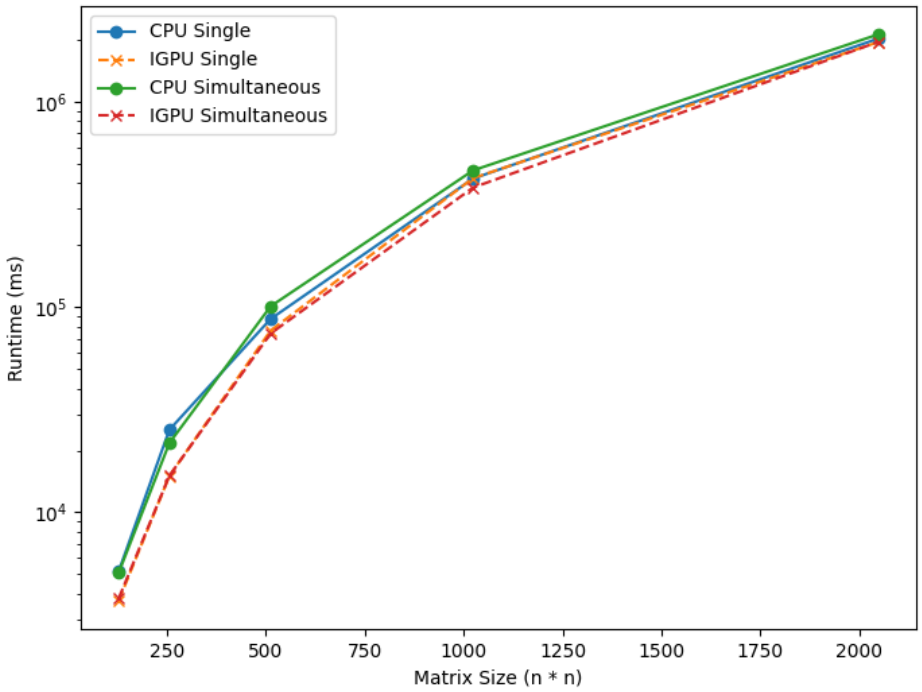

Figure 4.7: Comparison of SVD Kernel Runtimes on CPU and Integrated GPU: Simultaneous vs. Separate Execution.

Matrix Size	iGPU	CPU
128	3.811×10^3	5.084×10^3
256	1.5050×10^4	2.1714×10^4
512	7.4206×10^4	1.00616×10^5
1024	3.80871×10^5	4.61776×10^5
2048	1.949035×10^6	2.120034×10^6

Table 4.5: Simultaneous Execution Times for SVD Kernels on Intel(R) HD Graphics 520 and Intel(R) Core(TM) i5-6300U CPU

4.4. Performance Analysis of the 50% CPU / 50% iGPU Configuration

The runtimes presented were obtained from 5 iterations, and the average execution times were subsequently calculated for each case. As previously mentioned in the previous subsections, our sparse matrices consistently contain 20% of nonzero values. All the matrices, either sparse or dense, were created at runtime randomly. All time is expressed in milliseconds to make the comparison easier with the single device execution kernel times (already shown previously in subsection 4.2 in Figure 4.4 and Table 4.3).

As shown in Table 4.6 and Figure 4.8, the runtime increases dramatically with matrix size,

Matrix Size	50% CPU / 50% iGPU Time (ms)
128×128	2.34×10^4
256×256	1.14×10^5
512×512	6.04×10^5
1024×1024	2.53×10^6
2048×2048	1.08×10^7

Table 4.6: Runtime results for the 50/50 Jacobi scheduler

following a superlinear pattern. For instance, moving from a matrix size of 128×128 to 256×256 results in an almost five-times increase in runtime, while further increasing the matrix size to 512×512 leads to a similar jump. This exponential growth continues with larger matrices, where the runtime for a 2048×2048 matrix exceeds 2.78 hours, a drastic increase that reflects the high computational cost associated with handling large data volumes.

When comparing the $50/50$ scheduler to the CPU-only and iGPU-only configurations in Figure 4.9, it becomes apparent that the $50/50$ scheduler does not yield significant performance improvements over the iGPU-only configuration, especially for larger matrices. The iGPU-only configuration shows better runtimes, likely due to the iGPU's design, which is optimized for parallelized, data-intensive tasks. The current approach while intended to leverage both the CPU and iGPU, incurs additional overhead due to the fixed workload split and frequent synchronization between the two devices. This synchronization results in wait times as each device completes its portion of the task at each iteration, effectively reducing the potential performance benefits of parallel processing.

By enforcing a strict 50/50 workload split, the scheduler cannot adapt to the architectural differences between both devices, leading to suboptimal resource utilization. As the matrix size grows, the iGPU may struggle with tasks that require coordination with the CPU, while the CPU may become a bottleneck due to its limited parallel processing capabilities. This imbalance can contribute to the steeper increases in runtime seen in Table 4.6 and Figure 4.8.

Another significant factor affecting performance is the overhead of memory access and synchronization. Using *SYCL* ability to manage memory as needed allows CPU and iGPU to access shared data, but this introduces latency due to cache coherence and potential data transfer bottlenecks. With each iteration, the CPU and iGPU must synchronize their states, adding further delays. This synchronization overhead is impactful in the Jacobi method, where each cycle requires repeated coordination between devices. For larger

matrices, the time spent managing data consistency and ensuring synchronization adds substantially to the total runtime.

Our fixed implementation of $50/50$ split does not adjust based on real-time performance metrics, such as the current load on each device or variations in processing speed. This lack of adaptability prevents the scheduler from allocating tasks according to each device's capabilities or workload at any given moment. Consequently, there are periods when one device remains idle, waiting for the other to finish its assigned portion, stalling the potential gains from parallel execution.

In summary, the results shown in Table 4.6, Figure 4.8, and Figure 4.9 reveal that while the $50/50$ scheduler provides a straightforward approach to utilizing both CPU and iGPU resources, it suffers from several limitations. The increasing computational demands of larger matrices, coupled with synchronization overhead, shared memory access inefficiencies, and the lack of dynamic load balancing, contribute to the rapid increase in runtime as matrix size grows. To improve performance, future work could explore adaptive scheduling mechanisms that dynamically adjust the workload distribution based on device performance, thus optimizing resource utilization and potentially reducing the scaling inefficiencies identified in this analysis.

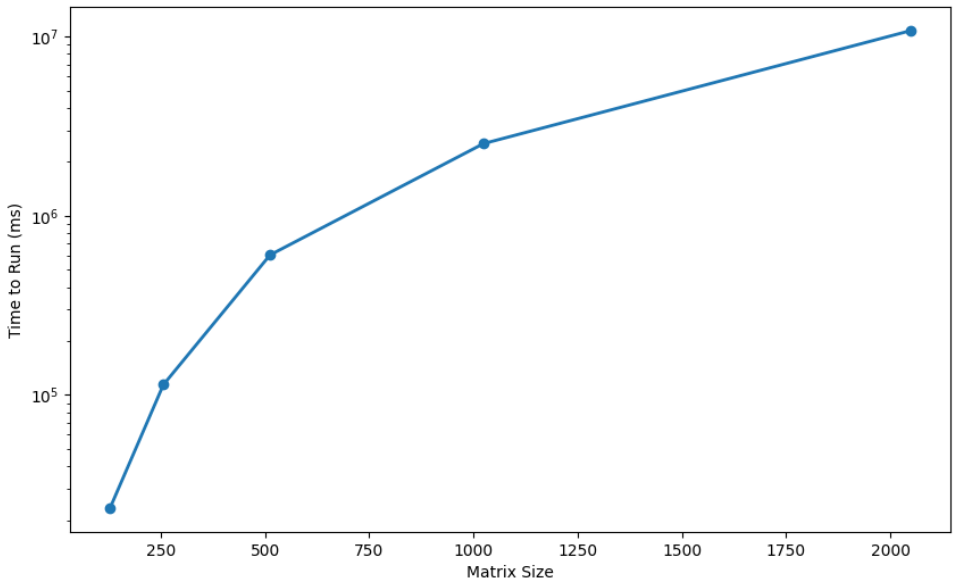

Figure 4.8: 50/50 iGPU/CPU Jacobi Scheduler Runtimes

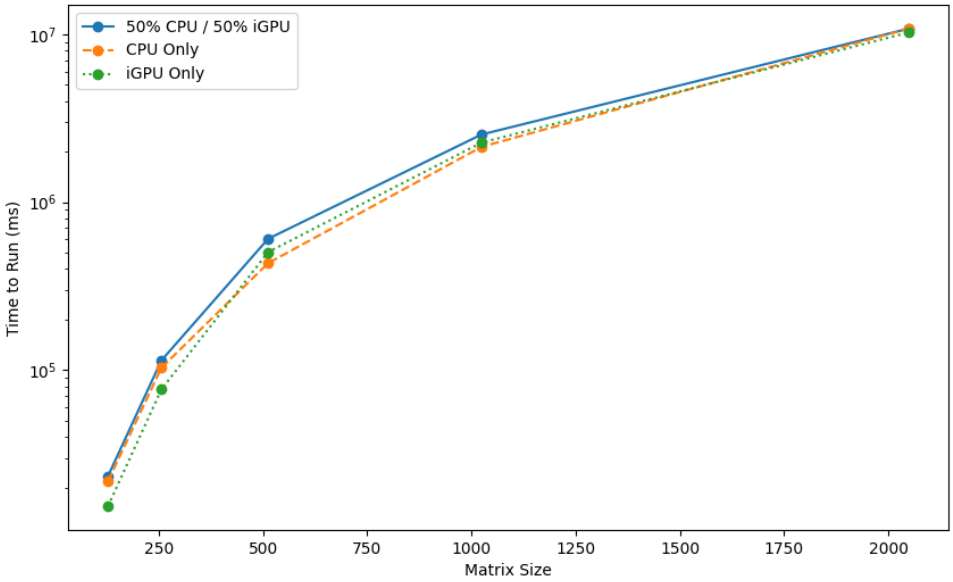

Figure 4.9: Scheduler compared to Single device Kernels.

5. Conclusions and Future Work

In this thesis, we explored the potential of heterogeneous computing systems by leveraging the combined processing power of multicore CPUs and integrated GPUs. The focus was primarily on two fundamental computational tasks, Sparse Matrix-Vector Multiplication (SpMV) and Singular Value Decomposition (SVD), widely used in scientific computing and data analysis. This work provided insights into the performance benefits and trade-offs associated with heterogeneous architectures by comparing different implementations and hardware setups.

The hybrid SVD implementation, which combined the Lanczos method for spectral decomposition with Jacobi rotations for diagonalization, exhibited promising results in execution time, particularly when offloaded to heterogeneous platforms. Additionally, the analysis of simultaneous execution across different hardware highlighted the importance of efficient task partitioning and workload balancing in heterogeneous computing environments.

The hybrid SVD implementation, which combined the Lanczos method for spectral decomposition with Jacobi rotations for diagonalization, showed better results in execution time. This implementation highlighted the advantages of parallelism and the capability of integrated GPUs to accelerate complex computations when used in conjunction with CPUs. Additionally, the analysis of simultaneous execution across different hardware underscored the importance of efficient task partitioning and workload balancing in heterogeneous computing environments.

A significant contribution of this study was the detailed analysis of a $50/50$ workload scheduler, which divided tasks equally between the CPU and iGPU. While the $50/50$ scheduler provided a straightforward approach to utilizing both processing units, it revealed several limitations, particularly as matrix size increased. Fixed workload partitioning, frequent synchronization requirements, and architectural differences between the CPU and iGPU led to performance bottlenecks, emphasizing the need for adaptive scheduling strategies. This finding highlights that a static split of tasks does not necessarily yield optimal performance, especially in heterogeneous systems where CPU and iGPU have distinct processing strengths.

The results underscore the potential of integrated GPUs to boost computational performance in scientific applications and reveal the critical role of flexible scheduling and load

balancing in fully exploiting the capabilities of heterogeneous architectures.

Future Work

A promising direction for future work is the development of a dynamic scheduler designed to optimize task distribution across CPU and integrated GPU resources. Unlike the static 50/50 approach, a dynamic scheduler could allocate tasks based on real-time performance metrics, adapting to the specific workload characteristics and the strengths of each processing unit. This would reduce idle times, minimize synchronization overhead, and improve resource utilization, particularly for complex workloads like SVD and SpMV.

Such an adaptive scheduler could incorporate workload-aware mechanisms that monitor CPU and iGPU performance in real time, redistributing tasks based on factors such as current load, thermal conditions, and task complexity. This approach would enhance the efficiency of heterogeneous computing systems, allowing them to handle larger and more complex workloads with improved scalability and responsiveness. Implementing these improvements could further unlock the potential of heterogeneous systems for high-performance scientific computing and data-intensive applications.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore: Johns Hopkins University Press, 1996.

[2] J. R. Gilbert, C. Moler, and R. Schreiber, “Sparse matrices in matlab: Design and implementation,” *SIAM J. Matrix Anal. Appl*, vol. 13, pp. 333–356, 1992.

[3] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[4] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 3rd ed. Berlin, New York: Springer-Verlag, 2002.

[5] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[6] R. Vuduc, J. Demmel, and K. Yelick, “Automatic performance tuning of sparse matrix-vector multiplications,” *ACM/IEEE conference on Supercomputing, SC’05*, pp. 1–25, 2005.

[7] P. Mpakos, D. Galanopoulos, P. Anastasiadis, N. Papadopoulou, N. Koziris, and G. Goumas, “Feature-based spmv performance analysis on contemporary devices,” 2023. [Online]. Available: https://arxiv.org/abs/2302.04225

[8] K. Akbudak, E. Kayaaslan, and C. Aykanat, “Hypergraph partitioning based models and methods for exploiting cache locality in sparse matrix-vector multiplication,” *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. C237–C262, 2013.

[9] M. Steinberger, A. Derlery, R. Zayer, and H.-P. Seidel, “How naive is naive spmv on the gpu?” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

[10] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 678–689.

[11] E. Kontoghiorghes, *Handbook of Parallel Computing and Statistics*, ser. Statistics: A Series of Textbooks and Monographs. CRC Press, 2005. [Online]. Available: https://books.google.li/books?id=BnNnKPkFH2kC

[12] C. Jacobi, “Über ein leichtes verfahren die in der theorie der säcularstörungen vorkommenden gleichungen numerisch aufzulösen.” *Journal für die reine und angewandte Mathematik*, vol. 30, pp. 51–94, 1846. [Online]. Available: http://eudml.org/doc/147275

[13] J. H. Wilkinson, “Note on the quadratic convergence of the cyclic jacobi process,” *Numerische Mathematik*, vol. 4, no. 1, pp. 296–300, 1962. [Online]. Available: https://doi.org/10.1007/BF01386321

[14] E. Kogbetliantz, “Solution of linear equations by diagonalization of coefficients matrix,” *Quarterly of Applied Mathematics*, vol. 13, pp. 123–132, 1955. [Online]. Available: http://www.ams.org/journals/qam/1955-13-02/S0033-569X-1955-88795-9/S0033-569X-1955-88795-9.pdf

[15] M. R. Hestenes, “Inversion of matrices by biorthogonalization and related results,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, pp. 51–90, 1958. [Online]. Available: https://doi.org/10.1137/0106005

[16] V. Hari and K. Veselic, “On the one-sided jacobi algorithm for singular value decomposition,” *Linear Algebra and its Applications*, vol. 338, pp. 113–130, 2001.

[17] K. K. Matam and K. Kothapalli, “Gpu accelerated lanczos algorithm with applications,” in *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, 2011, pp. 71–76.

[18] Z. Drmac and K. Veselic, “Jacobi methods for computing the singular value decomposition: A survey of computational derivations, implementations, and applications,” *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 4, pp. 1035–1062, 2008.

[19] Senheng, “Integrated vs dedicated graphics cards: What’s best for your laptop?” n.d., accessed: 2024-08-06. [Online]. Available: https://www.senheng.com.my/blog/beginner-guide/integrated-vs-dedicated-graphics-cards-laptop

[20] WikiChip, “Gen9.5 - microarchitectures - intel,” https://en.wikichip.org/wiki/intel/microarchitectures/gen9.5, 2018, accessed: 2024-08-06.

[21] Intel, *Intel Open Source HD Graphics and Intel Iris Plus Graphics Programmer’s Reference Manual*, January 2017.

[22] S. Junkins, “The compute architecture of intel processor graphics gen9,” Intel, August 2015.

[23] H. F. P. Duarte, “Performance analysis of intel gen9.5 integrated gpu architecture,” Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, June 2018.

[24] N. Corporation, “What is cuda?” https://nvidia.custhelp.com/app/answers/detail/a_id/2132/~/what-is-cuda%3F, accessed: 2024-08-06.

[25] ——, “Cuda toolkit,” https://developer.nvidia.com/cuda-toolkit, accessed: 2024-08-06.

[26] ——, “Cuda refresher: Cuda programming model,” https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/, accessed: 2024-08-06.

[27] K. Group, “Opencl,” https://www.khronos.org/opencl/, accessed: 2024-08-06.

[28] ——, “Opencl guide,” https://github.com/KhronosGroup/OpenCL-Guide, accessed: 2024-08-06.

[29] ——, “Sycl - the c++ single-source heterogeneous programming standard,” https://www.khronos.org/sycl/, accessed: 2024-08-06.

[30] Intel Corporation, “Sycl readme,” n.d., accessed: 2024-08-06. [Online]. Available: https://raw.githubusercontent.com/intel/llvm/sycl/README.md

[31] K. Group, “Sycl 2020 specification,” https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf, 2020, accessed: 2024-08-06.

[32] M. Costanzo, E. Rucci, C. García-Sánchez, M. Naiouf, and M. Prieto-Matías, “Comparing performance and portability between cuda and sycl for protein database search on nvidia, amd, and intel gpus,” in *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, Oct. 2023. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD59825.2023.00023

[33] M. Breyer, A. Van Craen, and D. Pflüger, “A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware,” in *Proceedings of the 10th International Workshop on OpenCL*, ser.

IWOCL '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3529538.3529980

[34] G. K. Reddy Kuncham, R. Vaidya, and M. Barve, "Performance study of gpu applications using sycl and cuda on tesla v100 gpu," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[35] R. R. Castro, "Sycl performance for nvidia and amd gpus matches native system language," https://www.oneapi.io/blog/sycl-performance-for-nvidia-and-amd-gpus-matches-native-system-language/, accessed: 2024-08-06.