Project Report

On

Optimization of Scientific Numerical Code using Parallel Programming

Submitted
In partial fulfilment

For the award of the Degree of

PG-Diploma in High Performance Computing Application Programming (PG-DHPCAP)

C-DAC, ACTS (Pune)

Guided By:

Dr. Nileshchandra Pikle

Submitted By:

Amar Ashok Badade (240840141002)

Rokade Sanket Vijay (240840141016)

Taseng Mancheykhun (240840141021)

Nishant kishor Ramteke (240840141009)

Vishal Sanjay Gujare (240840141022)

Centre for Development of Advanced Computing (C-DAC), ACTS (Pune-411008)

ABSTRACT

This project aims to optimize the performance of a sequential scientific simulation program by employing parallel programming techniques such as MPI, openMP, and CUDA for GPU acceleration. Using these techniques, the execution time is significantly reduced compared to the sequential implementation.

The optimization process focuses on key computational tasks, particularly matrix diagonalization, which is one of the most time-consuming steps in the simulation. GPU acceleration using CUDA's cuSolver library achieves a 36x reduction in computation time for this step. The benefits of MPI-based parallelization become more pronounced for large matrix sizes (above 6000×6000). Additionally, OpenMP enhances CPU-based parallelism by utilizing multi-threading, further improving overall efficiency. The hybrid approach combining CUDA, MPI, and OpenMP provides the most optimized performance, achieving a 2.8x overall speedup for a matrix size of 6500×6500 compared to the sequential program.

Table of Contents

Chapter 1

Introduction

1.1 Introduction

Scientific numerical simulation plays a crucial role in understanding complex physical, chemical, and engineering systems by solving mathematical models computationally. These simulations enable researchers to analyze real-world phenomena that are otherwise impractical or impossible to study experimentally, such as quantum mechanics, fluid dynamics, molecular interactions, climate modeling, and astrophysics. Numerical simulations often involve solving large-scale linear algebra problems, differential equations, and iterative computations, requiring substantial computational resources.

As scientific problems grow in complexity, optimizing numerical codes becomes essential to ensure faster execution, reduced memory usage, and efficient utilization of computational resources. Optimization techniques help in enhancing performance, reducing execution time, and enabling scalability across multi-core processors, clusters, and GPUs. Parallel programming plays a key role in this optimization process by distributing computations across multiple processing units, thus accelerating simulations that would otherwise take days or even weeks to complete.

1.2 Overview of the Simulation

In the project, we look at numerical solution of the Time-dependent Schrödinger Equation in the presence of a laser field. The Time-Dependent Schrödinger Equation (TDSE) is the fundamental equation governing the quantum evolution of a system over time. It describes how the quantum state of a particle or system evolves in response to external influences, such as electromagnetic fields. When a quantum system, such as an atom or molecule, is exposed to an intense laser field, its dynamics become highly nontrivial, leading to phenomena like photoionization, high-harmonic generation (HHG), and strong-field ionization. Mathematically, the TDSE for a single electron in an atomic potential under the influence of a laser field is given by:

$$i\hbar\frac{\partial}{\partial t}\Psi(r,t) = \hat{H}(t)\Psi(r,t)$$

where $\Psi(r,t)$ is the time-dependent wavefunction, and $\hat{H}(t)$ is the Hamiltonian operator of the system. In the presence of a laser field, the total Hamiltonian consists of two main components:

$$\hat{H}(t) = \hat{H}_0 + \hat{H}_{int}(t)$$

where:

$\hat{H}_0$ represents the field-free Hamiltonian, which describes the quantum system in the absence of an external field. For an atomic system, this is typically given by

$$\hat{H}_0 = \frac{p^2}{2m} + V(r)$$

where $\hat{p}$ is the momentum operator, m is the electron mass, and V(r) is the potential energy of the system (such as the Coulomb potential for an atom). $\hat{H}_{int}(t)$ represents the laser-atom interaction term, which depends on the form of the laser field coupling to the system. When a strong laser field is present, the interaction of the laser with the atomic or molecular system can be described using either the length gauge or the velocity gauge. The choice of gauge significantly affects the numerical implementation of the TDSE. In the length gauge representation,

$$\hat{H}_{int}(t) = e\vec{E}(t).\vec{r}$$

where e is the electron charge, and $\vec{E}(t)$ is the time-dependent electric field of the laser. In the velocity gauge representation,

$$\hat{H}_{int}(t) = \frac{e}{m}\vec{A}(t).\hat{p}$$

where $\vec{A}(t)$ is the vector potential of the laser field, related to the electric field $E(t) = -\frac{d}{dt}\vec{A}(t)$.

Solving the TDSE in the presence of a laser field is computationally challenging due to the high-dimensional nature of the wavefunction and the strong time dependence of the Hamiltonian. Several numerical methods are used for solving the TDSE. One such method is the Split-Operator Method. It is the most widely used method, which takes advantage of the Trotter approximation to evolve the wavefunction efficiently,

$$\Psi(r, t+\Delta t) = e^{-i\hat{H}\Delta t/\hbar}\Psi(r,t)$$

Using Trotter factorization, the evolution operator can be approximated as:

$$e^{-i\hat{H}\Delta t/\hbar} \approx e^{-i\hat{T}\Delta t/2\hbar} e^{-i\hat{V}\Delta t/\hbar} e^{-i\hat{T}\Delta t/2\hbar}$$

where $\hat{T}$ is the kinetic energy operator, and $\hat{V}$ is the potential energy operator.

This project focuses on the optimization of scientific numerical codes using parallel programming techniques such as OpenMP (for shared memory parallelism), MPI (for distributed computing), and CUDA (for GPU acceleration). In our project work, we parallelize a sequential C code that implements the simulation of time-propagation of a quantum mechanical wave function in a harmonic potential in the presence of a laser. The numerical operations in the program involves matrix operations, eigenvalue solving, and time evolution algorithm commonly used in quantum mechanics simulations. The key features of the code include constructing a Hamiltonian matrix, diagonalizing it, and propagating a wave function as initial state using matrix exponentials in the context of the Floquet formalism.

1.3 Objectives

The objectives of the project work are as follows –

- To parallelize the sequential numerical scientific simulation program.
- To apply parallelization techniques like MPI, openMP and CUDA to reduce the execution time of the sequential program.
- To compare the timings between sequential and parallel program.

Chapter 2
Literature Review

A Scientific Numerical simulation is one of the fundamental applications of High-Performance Computing (HPC). These simulations often require extensive computational resources to solve complex mathematical models, particularly in physics, chemistry, and engineering. A scientific numerical code typically involves solving large-scale mathematical problems, which frequently involve operations on high-dimensional matrices, such as matrix diagonalization, matrix multiplication, and eigenvalue problems. These computations are highly expensive in terms of execution time and memory consumption, making optimization techniques crucial for efficient execution.

In this project, we try to optimize a scientific numerical program that simulates a quantum particle evolving in time in the presence of a laser field. More specifically, we solve the time-dependent Schrodinger Equation (TDSE) in the presence of laser field. The numerical implementation employs a semi-classical approach where the system is represented quantum mechanically, while the laser field is treated classically.

2.1 Theoretical Background

We have a matrix called the Floquet Hamiltonian matrix $\mathbf{H}_F$ given as,

$$\mathbf{H}_F = \begin{bmatrix} \mathbf{H}_0 + 2\omega\mathbf{I} & (\epsilon_z\mathbf{Z})/2 & 0 & 0 & 0 \\ (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 + 1\omega\mathbf{I} & (\epsilon_z\mathbf{Z})/2 & 0 & 0 \\ 0 & (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 & (\epsilon_z\mathbf{Z})/2 & 0 \\ 0 & 0 & (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 - 1\omega\mathbf{I} & (\epsilon_z\mathbf{Z})/2 \\ 0 & 0 & 0 & (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 - 2\omega\mathbf{I} \end{bmatrix} \quad (1)$$

This Floquet matrix $\mathbf{H}_F$ is a supermatrix. A supermatrix is a large matrix that consists of smaller submatrices as its elements. We can see that $\mathbf{H}_F$ is in a structured block form, where each block is a matrix rather than a scalar.

The main equation involved in the simulation is to perform the operation of $e^{-\mathbf{H}_F\Delta t}$ on the initial wavefunction $\Psi(t_0)$, where t_0 is the initial time instant. To evolve the wavefunction $\Psi(t_0)$ in time by Δt, we perform,

$$e^{-\mathbf{H}_F\Delta t}\Psi(t_0 + \Delta t) = e^{-\mathbf{H}_F\Delta t}\Psi(t_0) \quad (2)$$

Now, it is not possible to take the exponential of the Floquet Hamitonian matrix, $\mathbf{H}_F$ unless we simplify the matrix. We take advantage of the fact that exponential of a diagonal matrix can be written as exponential of each individual elements as shown below,

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 & 0 & 0 & 0 \\ 0 & a_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 & 0 & 0 \\ 0 & 0 & 0 & a_4 & 0 \\ 0 & 0 & 0 & 0 & a_5 \end{bmatrix} \qquad e^{\mathbf{A}} = \begin{bmatrix} e^{a_1} & 0 & 0 & 0 & 0 \\ 0 & e^{a_1} & 0 & 0 & 0 \\ 0 & 0 & e^{a_1} & 0 & 0 \\ 0 & 0 & 0 & e^{a_1} & 0 \\ 0 & 0 & 0 & 0 & e^{a_1} \end{bmatrix} \quad (3)$$

The supermatrix $\mathbf{H}_F$ can be written as sum of supermatrices $\mathbf{H}_F^0$ and $\mathbf{H}_F^N$.

$$\mathbf{H}_F = \mathbf{H}_F^0 + \mathbf{H}_F^N$$

$$\mathbf{H}_F = \begin{bmatrix} \mathbf{H}_0 & (\epsilon_z\mathbf{Z})/2 & 0 & 0 & 0 \\ (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 & (\epsilon_z\mathbf{Z})/2 & 0 & 0 \\ 0 & (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 & (\epsilon_z\mathbf{Z})/2 & 0 \\ 0 & 0 & (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 & (\epsilon_z\mathbf{Z})/2 \\ 0 & 0 & 0 & (\epsilon_z\mathbf{Z})/2 & \mathbf{H}_0 \end{bmatrix} + \begin{bmatrix} 2\omega\mathbf{I} & 0 & 0 & 0 & 0 \\ 0 & 1\omega\mathbf{I} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1\omega\mathbf{I} & 0 \\ 0 & 0 & 0 & 0 & -2\omega\mathbf{I} \end{bmatrix} \quad (4)$$

$\mathbf{H}_F^N$ is a diagonal matrix. Its exponential can be taken using the method described in Equation (3). However, the exponential of $\mathbf{H}_F^0$ is tricky. We first need to diagonalize this supermatrix. This can be done using the following method, where $\mathbf{H}_F^0$ is simplified to $\mathbf{H}^{DM}$, a block diagonal supermatrix.

$$\mathbf{H}_F^0 = \mathbf{U}\mathbf{H}^{DM}\mathbf{U}^\dagger \quad (5)$$

$$\mathbf{H}_F^0 = \mathbf{U} \begin{bmatrix} \mathbf{H}_0 + \epsilon_2\mathbf{Z} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{H}_0 + \epsilon_2\mathbf{Z} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{H}_0 + \epsilon_3\mathbf{Z} & 0 & 0 \\ 0 & 0 & 0 & \mathbf{H}_0 + \epsilon_4\mathbf{Z} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{H}_0 + \epsilon_5\mathbf{Z} \end{bmatrix} \mathbf{U}^\dagger \quad (6)$$

where, $\mathbf{U}$ is called a unitary matrix and has the form,

$$U_{ij} = \sqrt{\frac{2}{N_F+1}}, i = 1, 2, 3..., N_F, j = 1, 2, 3..., N_F \quad (7)$$

where, N_F is the dimension of the supermatrix.

and

$$\epsilon_k = \epsilon_z cos(\tau_k), \text{where, } \tau_k = \left[\frac{k\pi}{N_F+1}\right], k = 1, 2, ..., N_F \quad (8)$$

Using these, we use a method called Split-Operator method and expand Eqn (2) to get,

$$e^{-\mathbf{H}_F\Delta t} \approx e^{-\mathbf{H}_F^N\Delta t/2}\mathbf{U}e^{-\mathbf{H}^{DM}\Delta t}\mathbf{U}^\dagger e^{-\mathbf{H}_F^N\Delta t/2} \quad (9)$$

where,

$$e^{-\mathbf{H}^{DM}\Delta t} = \begin{bmatrix} \mathbf{X}_1 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{X}_2 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{X}_3 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{X}_4 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{X}_5 \end{bmatrix} \quad (10)$$

and $$\mathbf{X}_k = e^{-i(\epsilon_k \mathbf{Z})\Delta t/2}\mathbf{U}_H e^{-iE\Delta t}\mathbf{U}_H^\dagger e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \quad (11)$$

where, k = [1, 5]

Chapter 3
Methodology and Implementation

3.1 Programming Language used in the program: C

C programming was used to write the program. C is one of the most widely used programming languages in scientific computing and high-performance computing (HPC) due to its efficiency, low-level memory control, and high portability. It is the language of choice for many numerical simulation applications, particularly in fields such as computational physics, chemistry, engineering, and machine learning. C provides direct hardware access, fine-tuned performance optimizations, and seamless integration with parallel computing frameworks like OpenMP, MPI, and CUDA. This article explores the role of C in numerical simulations and HPC, its usage with linear algebra packages like LAPACK and BLAS, its advantages over other languages, and how it compares with Fortran, another dominant language in scientific computing.

Scientific numerical simulations involve solving complex mathematical models, such as differential equations, matrix operations, and iterative methods, which require high computational power. In terms of performance and efficiency, C provides low-level memory control (e.g., pointers) that allows efficient memory management, reducing unnecessary overhead. Unlike high-level languages such as Python or MATLAB, C code is compiled directly into machine code, making it significantly faster. C is highly portable across different hardware architectures (x86, ARM, PowerPC, GPUs) and operating systems (Linux, macOS, Windows). Many scientific computing clusters and supercomputers run optimized C-based software. C can be easily integrated with Scientific Libraries and has seamless support for BLAS, LAPACK, and CUDA, which provide optimized mathematical routines for matrix and vector operations. Unlike Python or MATLAB, C does not require an interpreter or garbage collector, leading to faster execution times and reduced memory usage.

3.2 Libraries used in the program

LAPACK: LAPACK (Linear Algebra PACKage) is a standard software library for numerical linear algebra. It provides routines for solving systems of linear equations, linear least squares problems, eigenvalue problems, and singular value decomposition. LAPACK is widely used in scientific computing due to its efficiency and robustness.

BLAS: BLAS (Basic Linear Algebra Subprograms) is a collection of low-level routines for performing common linear algebra operations, such as vector and matrix multiplications, dot products, and vector norms. BLAS serves as the foundation for many high-performance libraries, including LAPACK, and is optimized for various hardware architectures.

cuSOLVER: cuSOLVER is an NVIDIA GPU-accelerated library for dense and sparse linear algebra. It provides optimized solvers for matrix factorization, eigenvalue problems, and linear system solutions. cuSOLVER is part of the NVIDIA CUDA toolkit and is designed to significantly accelerate scientific and engineering applications using GPU parallelism.

cuBLAS: cuBLAS (CUDA Basic Linear Algebra Subprograms) is NVIDIA's GPU-accelerated library for performing high-performance linear algebra operations. It offers optimized implementations of BLAS routines for CUDA-enabled GPUs, enabling efficient execution of matrix multiplications, vector operations, and other essential linear algebra computations. cuBLAS is extensively used in scientific computing, engineering simulations, and deep learning applications.

3.3 Parallelization techniques used

MPI: MPI (Message Passing Interface) is a widely used parallel programming model for distributed-memory systems. It enables communication between multiple processes running on different computing nodes. MPI is employed in this program to distribute computational tasks across multiple processors, allowing for efficient large-scale simulations and data processing.

openMP: OpenMP (Open Multi-Processing) is an API that supports shared-memory parallel programming in C, C++, and Fortran. It allows the parallel execution of loops and sections of code using compiler directives. OpenMP is used in this program to exploit multi-core CPU architectures by dividing computational workloads among multiple threads, improving execution speed and resource utilization.

CUDA: CUDA (Compute Unified Device Architecture) is a parallel computing platform and API developed by NVIDIA for programming GPUs. It enables massive parallelism by allowing thousands of threads to execute simultaneously. CUDA is used in this program to accelerate matrix computations and linear algebra operations, particularly through libraries like cuBLAS and cuSOLVER, significantly improving performance for large-scale scientific simulations.

3.4 The Sequential program

A sequential program executes instructions in the order they're written. Each instruction must be completed before the next one can begin. The program waits for expected events in the execution path.

In Fig.1, a flowchart of the algorithm for the sequential version of the program is shown. It starts with a Hamiltonian matrix which is created from the Kinetic and Potential energy matrix in the Discrete Variable Representation (DVR) basis. The potential energy function is a Harmonic oscillator function. It represents a quantum particle in a harmonic potential well. Our aim is to simulate the behavior of the quantum particle in the presence of a time-varying laser field represented by $E(t) = \epsilon_o cos(\omega t)$.

The Hamiltonian matrix is diagonalized to obtain the eigenvectors and corresponding eigenvalues. The matrix diagonalization is performed using LAPACKE_dsyev function from the LAPACK library. After we obtain the eigenvectors of the Hamiltonian matrix, one of the eigenstate is chosen to represent the intitial wavefunction of the quantum particle. The intermediate step of constructing the matrix 'exhm' involves multiplication of complex matrices, which is performed using cblas_zgemm function from the BLAS library. After this, we start the time loop. Each loop propagates the wavefunction upto a time step Δt. Inside the loop, we first perform $e^{-\mathbf{H}_F^N \Delta t/2}\Psi(t_0)$. Here, $e^{-\mathbf{H}_F^N \Delta t/2}$ is a square matrix and $\Psi(t_0)$ is a column matrix. Multiplication of both gives us a column matrix say A. Here, the matrix multiplication happens index wise, so we don't use BLAS function for the matrix multiplication. Next, A is right multiplied to $e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times \mathbf{U}^\dagger$ to obtain column vector B. Next, B is right multiplied to the square matrix 'exhm' that we created before the start of the loop. Here, again we obtain a column matrix say C. In this step, we use BLAS function cblas_zgemm to perform the complex matrix multiplication. The column matrix C is further right multiplied to $\mathbf{U} \times e^{-i(\epsilon_k \mathbf{Z})\Delta t/2}$ to obtain column vector D. Furthermore, D is right multiplied to $e^{-\mathbf{H}_F^N \Delta t/2}$ to obtain the wavefunction

$\Psi(t_0 + \Delta t)$ that has evolved in time by Δt. Lastly, we extract a quantum mechanical property of the quantum particle in a laser field called the induced dipole moment. The evolved wavefunction in this iteration of the loop is given as input to the next iteration of the loop. At every iteration, which means at every time instant, we extract the induced dipole moment of the quantum particle in the laser field.

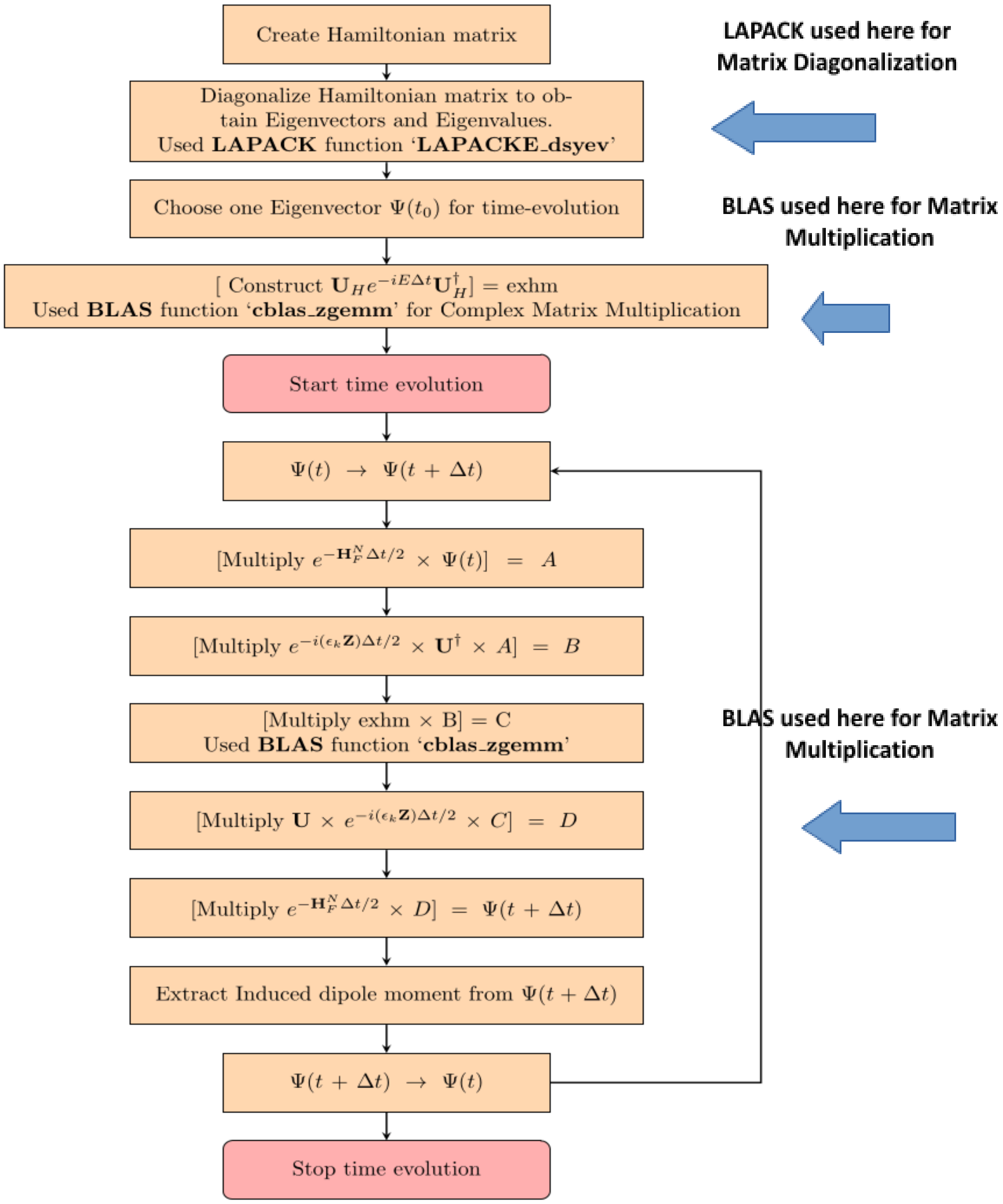
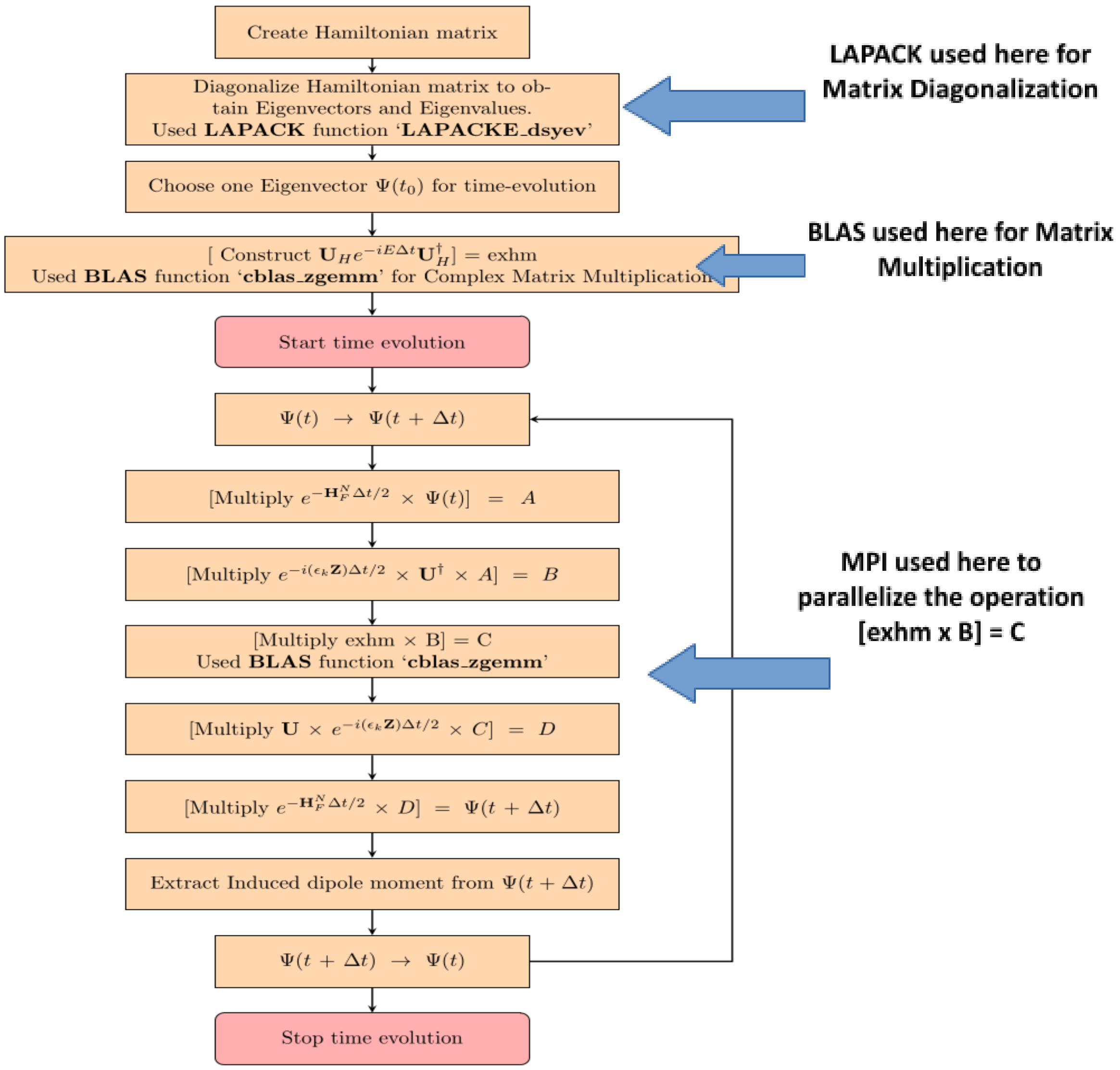

Create Hamiltonian matrix
Diagonalize Hamiltonian matrix to obtain Eigenvectors and Eigenvalues. Used LAPACK function 'LAPACKE_dsyev'
Choose one Eigenvector $\Psi(t_0)$ for time-evolution
[Construct $\mathbf{U}_H e^{-iE\Delta t}\mathbf{U}_H^\dagger$] = exhm Used BLAS function 'cblas_zgemm' for Complex Matrix Multiplication
Start time evolution
$\Psi(t) \rightarrow \Psi(t+\Delta t)$
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times \Psi(t)$] = A
[Multiply $e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times \mathbf{U}^\dagger \times A$] = B
[Multiply exhm × B] = C Used BLAS function 'cblas_zgemm'
[Multiply $\mathbf{U} \times e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times C$] = D
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times D$] = $\Psi(t+\Delta t)$
Extract Induced dipole moment from $\Psi(t+\Delta t)$
$\Psi(t+\Delta t) \rightarrow \Psi(t)$
Stop time evolution
LAPACK used here for Matrix Diagonalization
BLAS used here for Matrix Multiplication
BLAS used here for Matrix Multiplication

Fig. 3.1: Flowchart of the sequential version of the program

3.5 Parallelization using MPI

The algorithm of the parallel code is represented using the flowchart as shown in Fig.2. We apply MPI parallelization technique to parallelize the matrix multiplication exhm x B = C inside the time loop.



Create Hamiltonian matrix
Diagonalize Hamiltonian matrix to obtain Eigenvectors and Eigenvalues. Used **LAPACK** function '**LAPACKE_dsyev**'
LAPACK used here for Matrix Diagonalization
Choose one Eigenvector $\Psi(t_0)$ for time-evolution
[Construct $\mathbf{U}_H e^{-iE\Delta t}\mathbf{U}_H^\dagger$] = exhm
Used **BLAS** function '**cblas_zgemm**' for Complex Matrix Multiplication
BLAS used here for Matrix Multiplication
Start time evolution
$\Psi(t) \rightarrow \Psi(t+\Delta t)$
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times \Psi(t)$] $= A$
[Multiply $e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times \mathbf{U}^\dagger \times A$] $= B$
[Multiply exhm × B] = C
Used **BLAS** function '**cblas_zgemm**'
MPI used here to parallelize the operation [exhm x B] = C
[Multiply $\mathbf{U} \times e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times C$] $= D$
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times D$] $= \Psi(t+\Delta t)$
Extract Induced dipole moment from $\Psi(t+\Delta t)$
$\Psi(t+\Delta t) \rightarrow \Psi(t)$
Stop time evolution

Fig. 3.2: Flowchart of the parallel version of the program using MPI

In each time instant, the master process with rank=0 sends the square matrix 'exhm' to other worker processes of rank$\neq$0. This step of matrix multiplication is parallelized because earlier from eqn (9), we find that,

$$e^{-\mathbf{H}_F \Delta t} \approx e^{-\mathbf{H}_F^N \Delta t/2} \mathbf{U} e^{-\mathbf{H}^{DM} \Delta t} \mathbf{U}^\dagger e^{-\mathbf{H}_F^N \Delta t/2}$$

where,

$$e^{-\mathbf{H}^{DM}\Delta t} = \begin{bmatrix} \mathbf{X}_1 & 0 & 0 & 0 & 0 \\ 0 & \mathbf{X}_2 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{X}_3 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{X}_4 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{X}_5 \end{bmatrix}$$

and $\mathbf{X}_k = e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \mathbf{U}_H e^{-iE\Delta t} \mathbf{U}_H^\dagger e^{-i(\epsilon_k \mathbf{Z})\Delta t/2}$ where, k = [1, 5]

Here, $\mathbf{U}_H e^{-iE\Delta t} \mathbf{U}_H^\dagger$ = exhm. Since this matrix 'exhm' is common in all diagonal block elements in $e^{-\mathbf{H}^{DM}\Delta t}$, we send the matrix 'exhm' and the particular section of the column vector 'B' of dimension same as that of exhm to other processes of rank$\neq$0 and perform the matrix-vector multiplication there. This is implemented using functions MPI_Send and MPI_Recv of MPI. We also comment that the wavefunction in each successive time loop is dependent on preceeding wavefunction, hence parallelization of the loop is not possible.

3.6 Parallelization using MPI + CUDA

Fig.3 shows the flowchart of the algorithm of parallelization using MPI with added parallelization using CUDA. From the analysis of previous parallel versions of the program, we observed that the matrix diagonalization step is the most expensive step. So, it was obvious to aim to reduce execution time of this step. We use CUDA's cuSOLVER library, from which we use 'cusolverDnDsyevd' function to diagonalize the symmetric matrix. It was observed that this reduced the matrix diagonalization execution time by about 36 times.

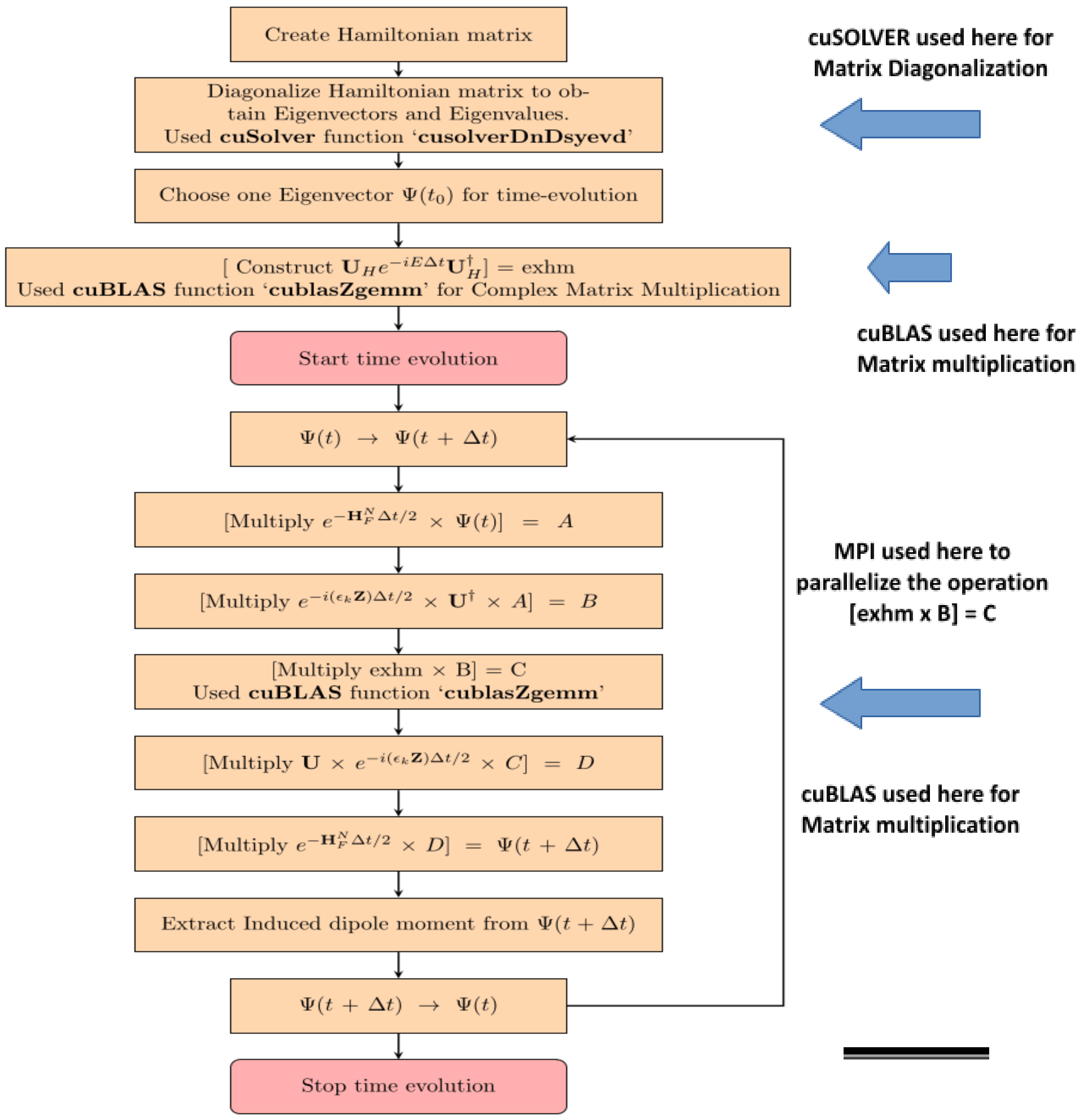
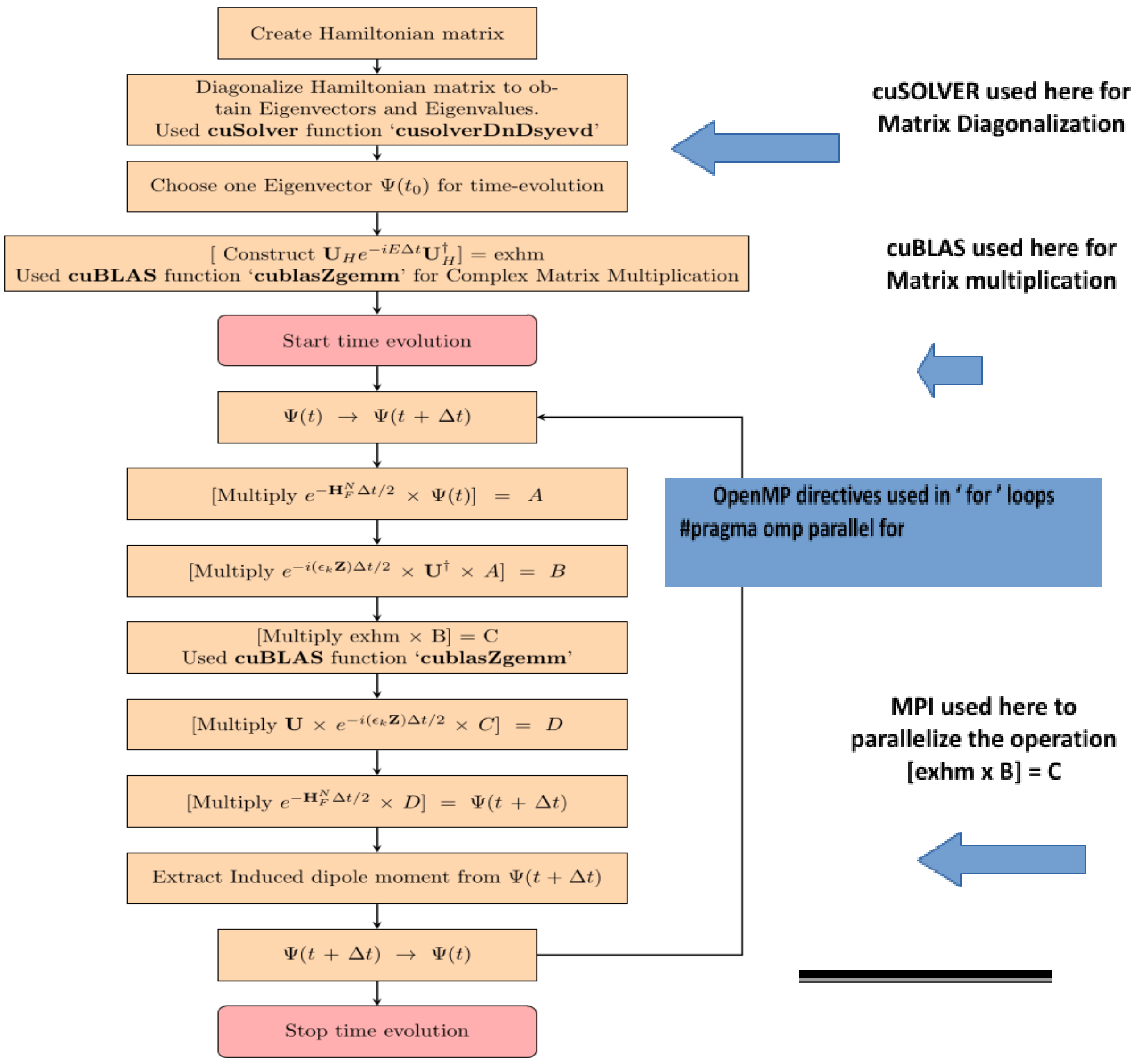

Create Hamiltonian matrix
Diagonalize Hamiltonian matrix to obtain Eigenvectors and Eigenvalues. Used **cuSolver** function '**cusolverDnDsyevd**'
Choose one Eigenvector $\Psi(t_0)$ for time-evolution
[Construct $\mathbf{U}_H e^{-iE\Delta t}\mathbf{U}_H^\dagger$] = exhm
Used **cuBLAS** function '**cublasZgemm**' for Complex Matrix Multiplication
Start time evolution
$\Psi(t) \rightarrow \Psi(t+\Delta t)$
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times \Psi(t)$] $= A$
[Multiply $e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times \mathbf{U}^\dagger \times A$] $= B$
[Multiply exhm × B] = C
Used **cuBLAS** function '**cublasZgemm**'
[Multiply $\mathbf{U} \times e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times C$] $= D$
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times D$] $= \Psi(t+\Delta t)$
Extract Induced dipole moment from $\Psi(t+\Delta t)$
$\Psi(t+\Delta t) \rightarrow \Psi(t)$
Stop time evolution
cuSOLVER used here for Matrix Diagonalization
cuBLAS used here for Matrix multiplication
MPI used here to parallelize the operation [exhm x B] = C
cuBLAS used here for Matrix multiplication

Fig. 3.3: Flowchart of the parallel version of the program using MPI + CUDA

We also use CUDA’s cuBLAS library, from which we use ‘cublasZgemm’ function to perform complex valued matrix multiplication. cuSOLVER and cuBLAS work with column-major order, so in C, where row-major order is used, one must transpose and also flatten the matrix, so that it is compatible to be used for cuSOLVER and cuBLAS library.

3.7 Parallelization using MPI + CUDA + openMP


Create Hamiltonian matrix
Diagonalize Hamiltonian matrix to obtain Eigenvectors and Eigenvalues. Used **cuSolver** function ‘**cusolverDnDsyevd**’
cuSOLVER used here for Matrix Diagonalization
Choose one Eigenvector $\Psi(t_0)$ for time-evolution
[Construct $\mathbf{U}_H e^{-iE\Delta t}\mathbf{U}_H^\dagger$] = exhm Used **cuBLAS** function ‘**cublasZgemm**’ for Complex Matrix Multiplication
cuBLAS used here for Matrix multiplication
Start time evolution
$\Psi(t) \rightarrow \Psi(t+\Delta t)$
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times \Psi(t)$] = A
OpenMP directives used in ‘ for ’ loops
#pragma omp parallel for
[Multiply $e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times \mathbf{U}^\dagger \times A$] = B
[Multiply exhm × B] = C Used **cuBLAS** function ‘**cublasZgemm**’
[Multiply $\mathbf{U} \times e^{-i(\epsilon_k \mathbf{Z})\Delta t/2} \times C$] = D
MPI used here to parallelize the operation [exhm x B] = C
[Multiply $e^{-\mathbf{H}_F^N \Delta t/2} \times D$] = $\Psi(t+\Delta t)$
Extract Induced dipole moment from $\Psi(t+\Delta t)$
$\Psi(t+\Delta t) \rightarrow \Psi(t)$
Stop time evolution

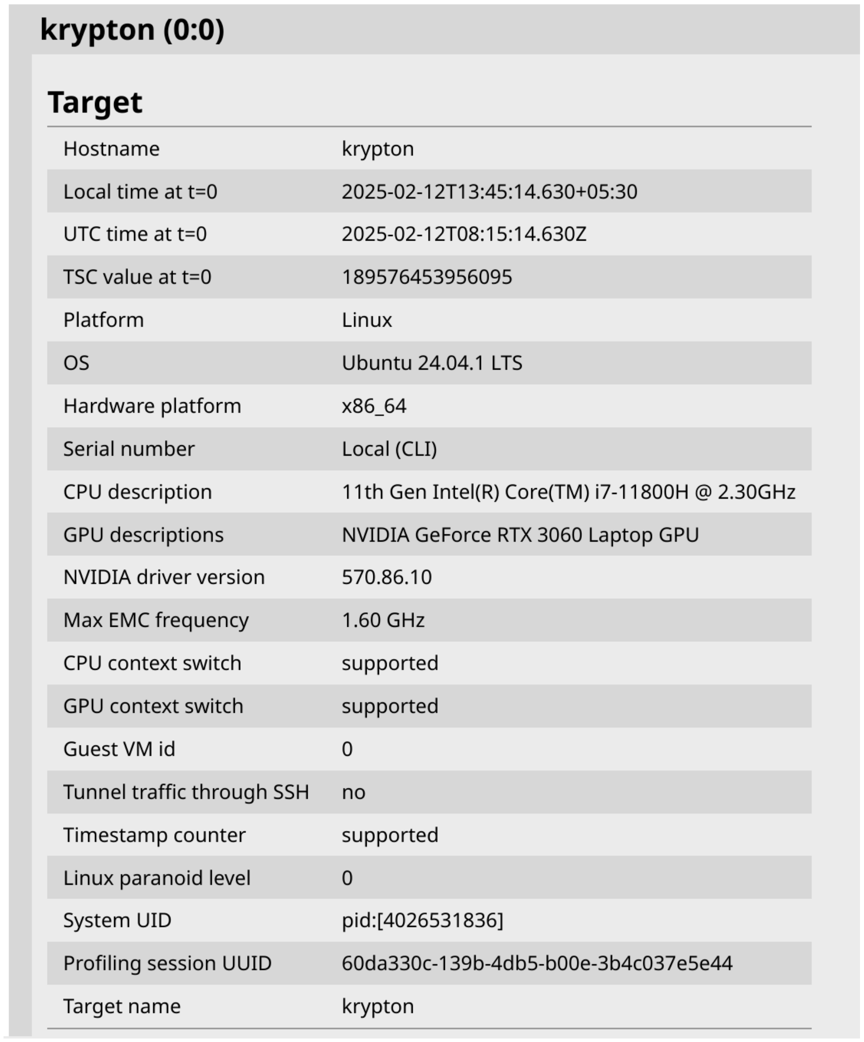
cuBLAS used here for
Matrix multiplication

Fig. 3.4: Flowchart of the parallel program using MPI + CUDA+openMP

Fig.4 shows the implementation of the parallel version of the code that contains MPI, CUDA and openMP. We use pragma directives to create parallel regions in the for loops in the program using the ‘for’ construct. We have fixed the total number of threads to 50. This is because, we observed the most optimized execution time was obtained in this thread number. We use the construct ‘collapse’ for 2 nested for loops. We also use ‘reduction’ clause to perform parallel aggregate sum operations keeping in mind the shared and private variables, hence avoiding race condition.

3.8 Hardware Configurations:

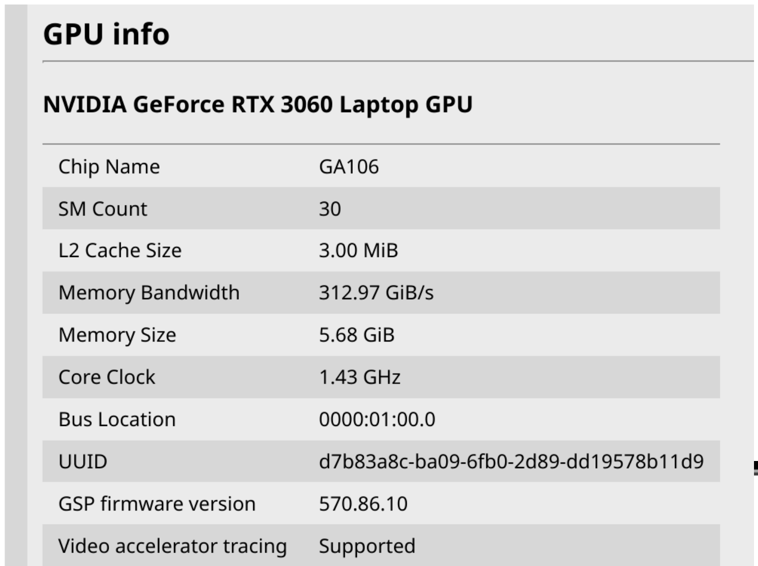
- Platform: Linux
- OS: Ubuntu 24.04.1 LTS
- OS Type: 64-bit
- CPU description: 11th Gen Intel® Core™ i7-11800H × 16
- Number of Cores: 8
- Clock Speed: 2.30 GHz
- RAM: 16 GB
- RAM Type: DDR4
- RAM Frequency: 3200 MHz
- Cache: 24 MB
- GPU description: NVIDIA GeForce RTX™ 3060 Laptop GPU
- Dedicated Graphics Memory Type: GDDR6
- Dedicated Graphic Memory Capacity: 6 GB



krypton (0:0)

Target

Hostname	krypton
Local time at t=0	2025-02-12T13:45:14.630+05:30
UTC time at t=0	2025-02-12T08:15:14.630Z
TSC value at t=0	189576453956095
Platform	Linux
OS	Ubuntu 24.04.1 LTS
Hardware platform	x86_64
Serial number	Local (CLI)
CPU description	11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
GPU descriptions	NVIDIA GeForce RTX 3060 Laptop GPU
NVIDIA driver version	570.86.10
Max EMC frequency	1.60 GHz
CPU context switch	supported
GPU context switch	supported
Guest VM id	0
Tunnel traffic through SSH	no
Timestamp counter	supported
Linux paranoid level	0
System UID	pid:[4026531836]
Profiling session UUID	60da330c-139b-4db5-b00e-3b4c037e5e44
Target name	krypton



GPU info

NVIDIA GeForce RTX 3060 Laptop GPU

Chip Name	GA106
SM Count	30
L2 Cache Size	3.00 MiB
Memory Bandwidth	312.97 GiB/s
Memory Size	5.68 GiB
Core Clock	1.43 GHz
Bus Location	0000:01:00.0
UUID	d7b83a8c-ba09-6fb0-2d89-dd19578b11d9
GSP firmware version	570.86.10
Video accelerator tracing	Supported

Chapter 4
Results

We have developed three different versions of the parallel program to analyze their performance during the simulation. These versions include a sequential implementation that uses LAPACK and BLAS function, an MPI-based parallel implementation that uses LAPACK and BLAS, a hybrid parallel implementation that uses MPI and GPU-accelerated implementation that uses cuSOLVER and cuBLAS, and another hybrid parallel implementation that uses MPI, GPU-accelerated implementation using cuSOLVER and cuBLAS and openMP. To evaluate the performance of each version, we executed them on matrices of varying sizes, ranging from 500 to 7000 for the block matrix within the supermatrix. The primary goal of these analysis is to assess the efficiency of different diagonalization approaches, different matrix multiplication approaches and determine the impact of parallelization on execution time. By systematically increasing the matrix size, we gain insights into how each approach scales and how communication overhead, computational workload distribution, and GPU acceleration affect the overall performance of the simulation.

4.1 Optimization of Matrix Diagonalization

In all versions of the program, matrix diagonalization is a crucial computational step. Here, we compare the diagonalization using LAPACK's function in sequential program, using LAPACK's function in MPI program and using cuSOLVER's function in MPI program. Surprisingly, we observe that the diagonalization step in MPI implementation is slower than that of sequential version even though both are using the same LAPACK function. This performance degradation can be likely due to communication overload in the MPI implementation, which offsets the benefits the benefits of parallel execution. However, in the GPU-accelerated version, where we replace LAPACK with cuSOLVER's function for matrix diagonalization, we achieve a 36x speedup compared to the sequential implementation. This significant performance improvement highlights the efficiency of GPU-based diagonalization and demonstrates the advantages of using CUDA-accelerated libraries for large-scale matrix operations.

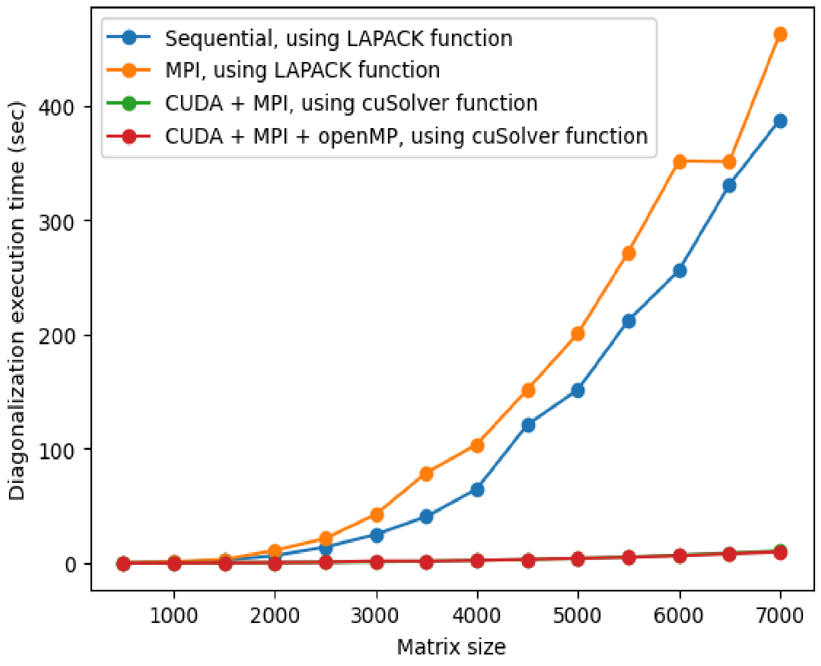


Fig. 4.1: Comparison of execution time of matrix diagonalization

4.2 Optimization of Matrix Multiplication

The program involves multiple matrix multiplication operations, which are among the most computationally intensive tasks. As a result, it is crucial to analyze and compare the execution efficiency of matrix multiplication across different parallelized implementations. Understanding how each approach handles these operations helps in identifying bottlenecks and optimizing performance.

As shown in Figure 4.2, an unexpected observation is that the sequential version, which utilizes BLAS for matrix multiplication, outperforms all the parallel implementations, including the cuBLAS-based GPU-accelerated version. This suggests that the optimized BLAS routines in the sequential program are highly efficient for the given workload.

Despite using the same BLAS functions for matrix multiplication in both the sequential and MPI-based parallel implementations, we observe a noticeable performance drop in the MPI version. This degradation can likely be attributed to communication overhead introduced by MPI. In the MPI implementation, matrices need to be distributed across processes, and frequent inter-process communication may lead to latency and synchronization delays, reducing overall efficiency. Additionally, splitting matrices into blocks for distributed computation may result in non-optimal memory access patterns, further affecting performance.

This analysis highlights that while parallelization can improve scalability, it does not always guarantee superior performance, especially when communication costs outweigh computational benefits.

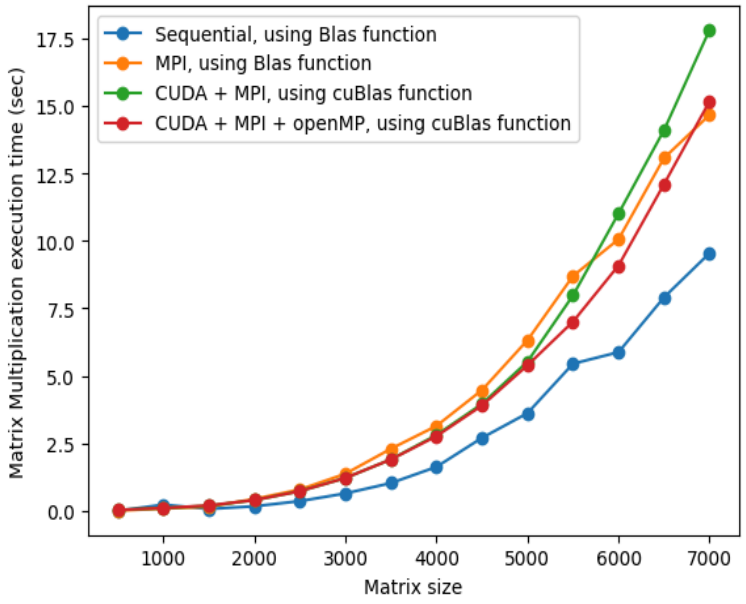


Fig. 4.2: Comparison of execution time of matrix multiplication

4.3 Optimization of loop execution time

The time loop is the most crucial part of the program, as it is where the main simulation takes place, evolving the system over discrete time steps. Given its significance, it is essential to analyze and compare the execution performance of the time loop across all parallel implementations. To ensure a fair comparison, we calculate the average execution time per iteration and use this as a metric to evaluate the efficiency of each parallel version.

From our observations, the CUDA + MPI + OpenMP implementation emerges as the most optimized version. Interestingly, despite using the same cuBLAS functions as the CUDA + MPI implementation, the hybrid approach incorporating OpenMP achieves a more efficient execution. This suggests that multi-threading on CPUs (via OpenMP) alongside GPU acceleration (via cuBLAS) leads to better resource utilization and improved overall performance.

Another key observation is that the standalone MPI implementation is the most computationally expensive for smaller matrix sizes (e.g., 500). However, as the matrix size increases up to 6000, the execution time for the MPI version improves significantly compared to the other parallel implementations. This can be attributed to the way matrix multiplications are handled within the MPI implementation—instead of multiplying the entire supermatrix at once, computations are performed on block matrices, leading to better memory management and efficiency as problem size scales.

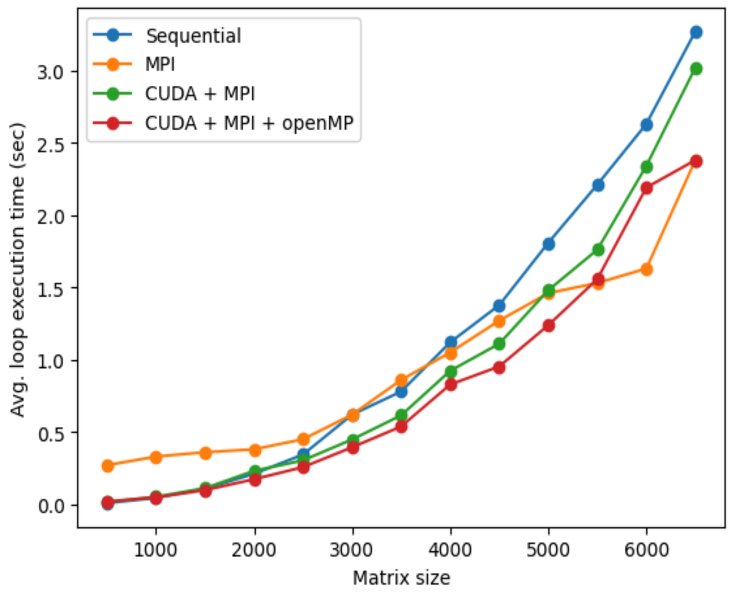


Fig. 4.3: Comparison of loop execution time

Additionally, the combined use of cuBLAS for GPU-accelerated matrix operations and OpenMP for parallelizing smaller for-loops further contributes to the overall speedup of the program. By efficiently distributing the computational workload across both GPUs and multi-core CPUs, the hybrid approach effectively minimizes execution time.

In our simulation, the time loop executes 62 iterations, a value determined by the number of optical cycles and the frequency of the oscillating laser field. Since this loop constitutes the majority of the program's runtime, optimizing it is crucial for enhancing the overall performance of the simulation.

4.4 Optimization of total execution time

Finally, it is essential to compare the total execution time of all three parallel implementations to determine their efficiency and scalability. As shown in Figure 4.4, we observe that the hybrid implementation (CUDA + MPI + OpenMP) is the most optimized parallel version among the three. When compared to the sequential implementation, this hybrid approach achieves a 2.8x speedup, demonstrating its effectiveness in handling large matrix computations.

When we compare the CUDA + MPI implementation with the CUDA + MPI + OpenMP implementation, we notice an interesting trend. For smaller matrix sizes, both implementations exhibit similar execution times, indicating that the additional OpenMP parallelization does not provide a significant advantage at this scale. However, as the matrix size increases, the CUDA + MPI + OpenMP approach consistently outperforms the CUDA + MPI version, completing the computations in less time. This suggests that OpenMP effectively utilizes multiple threads and improves performance for larger matrices.

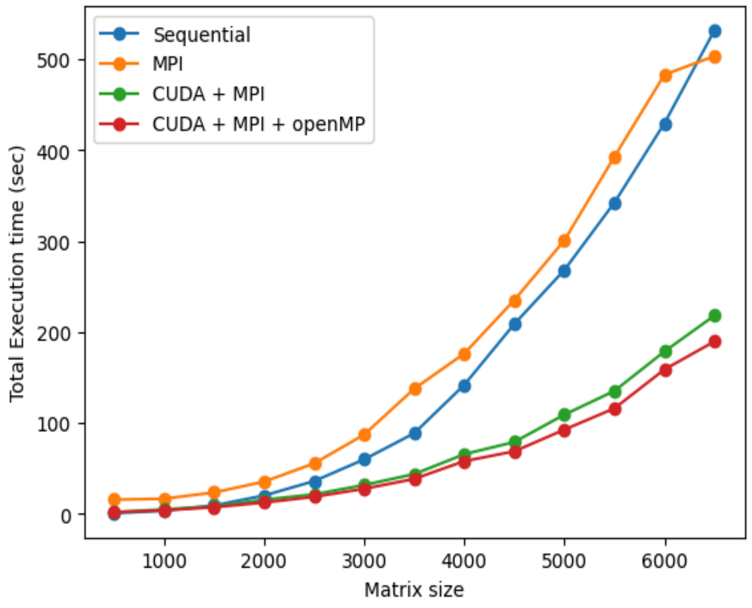


Fig. 4.4: Comparison of total execution time of the program

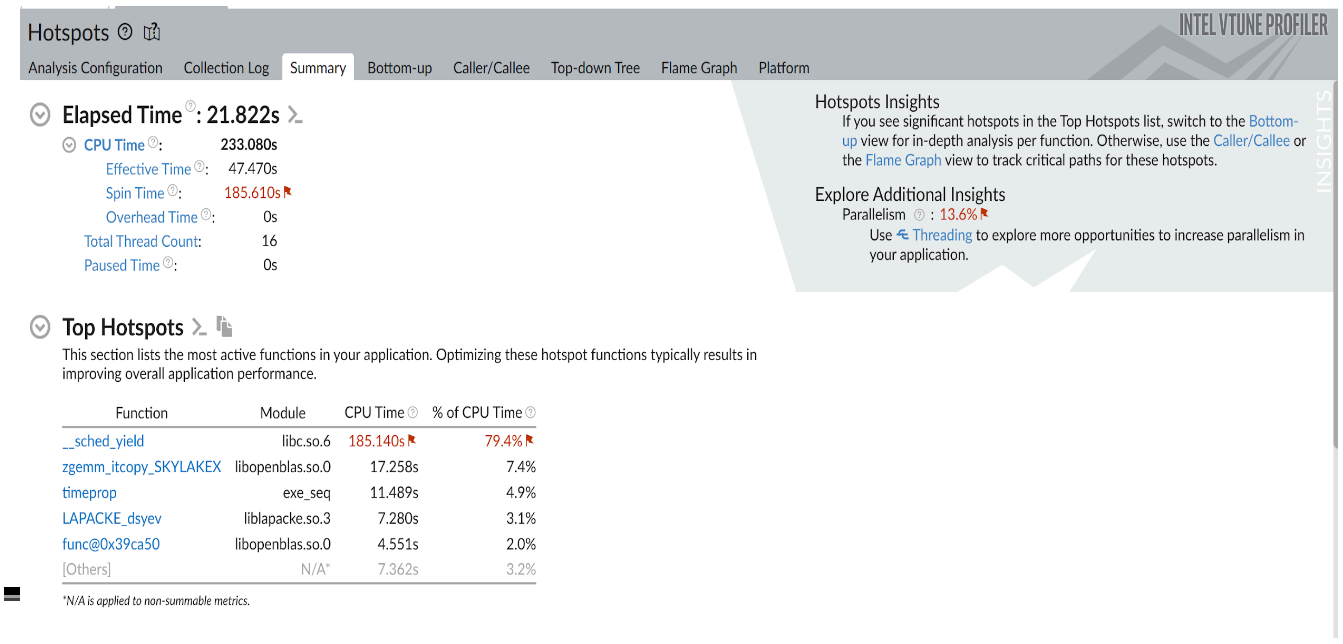
Additionally, we observe that the sequential implementation executes faster than the MPI-only implementation, which seems counterintuitive. However, this can be attributed to communication overhead in the MPI-based approach, where frequent inter-process communication and data exchange negatively impact performance, especially for the given workload. This further highlights the importance of choosing an appropriate parallelization strategy based on problem size and architecture.

Overall, our results demonstrate that while CUDA and MPI provide substantial acceleration, the addition of OpenMP in a hybrid parallelization model enhances performance further, making it the most efficient approach for large-scale matrix computations.

4.5 Profiling reports

4.5.1 Sequential code profiling

Software used: Intel's Vtune Profiler



Elapsed Time: 21.822s

CPU Time:	233.080s
Effective Time:	47.470s
Spin Time:	185.610s
Overhead Time:	0s
Total Thread Count:	16
Paused Time:	0s

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism : 13.6%

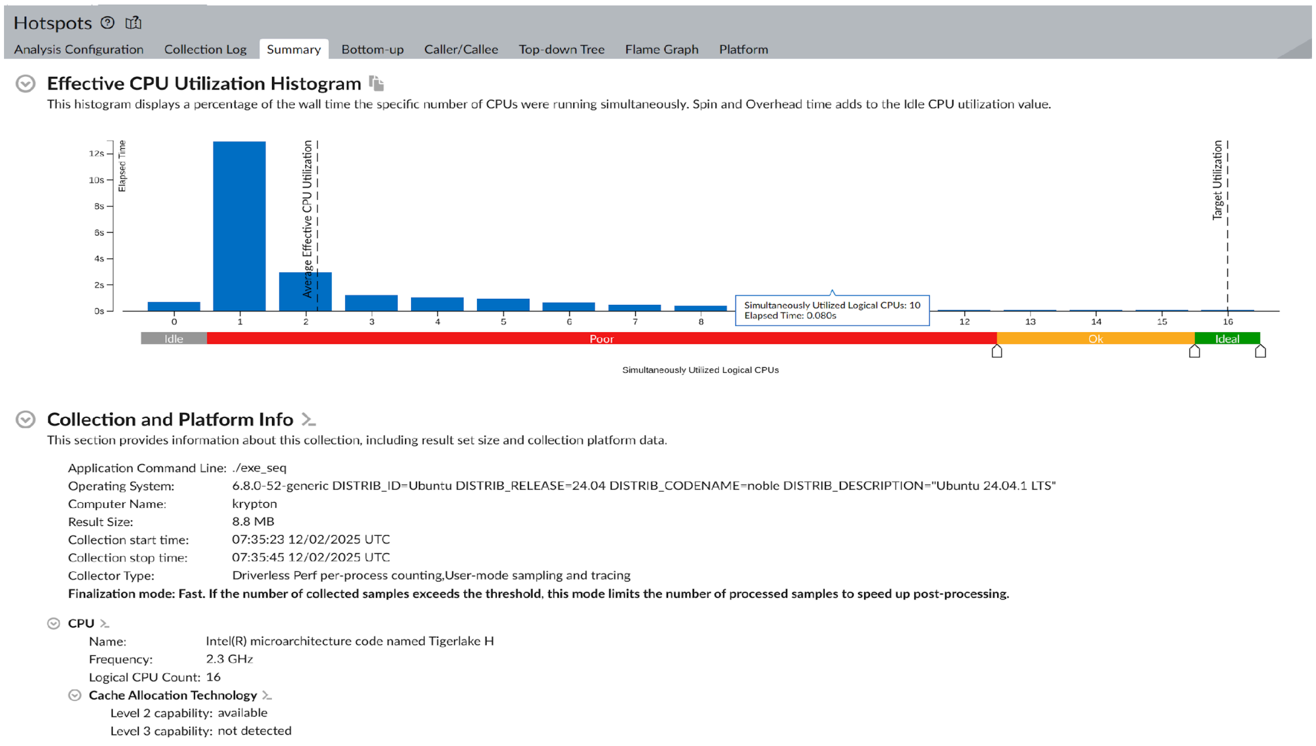
Use Threading to explore more opportunities to increase parallelism in your application.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
__sched_yield	libc.so.6	185.140s	79.4%
zgemm_itcopy_SKYLAKEX	libopenblas.so.0	17.258s	7.4%
timeprop	exe_seq	11.489s	4.9%
LAPACKE_dsyev	liblapacke.so.3	7.280s	3.1%
func@0x39ca50	libopenblas.so.0	4.551s	2.0%
[Others]	N/A*	7.362s	3.2%

*N/A is applied to non-summable metrics.



Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line:	./exe_seq
Operating System:	6.8.0-52-generic DISTRIB_ID=Ubuntu DISTRIB_RELEASE=24.04 DISTRIB_CODENAME=noble DISTRIB_DESCRIPTION="Ubuntu 24.04.1 LTS"
Computer Name:	krypton
Result Size:	8.8 MB
Collection start time:	07:35:23 12/02/2025 UTC
Collection stop time:	07:35:45 12/02/2025 UTC
Collector Type:	Driverless Perf per-process counting,User-mode sampling and tracing

Finalization mode: Fast. If the number of collected samples exceeds the threshold, this mode limits the number of processed samples to speed up post-processing.

CPU

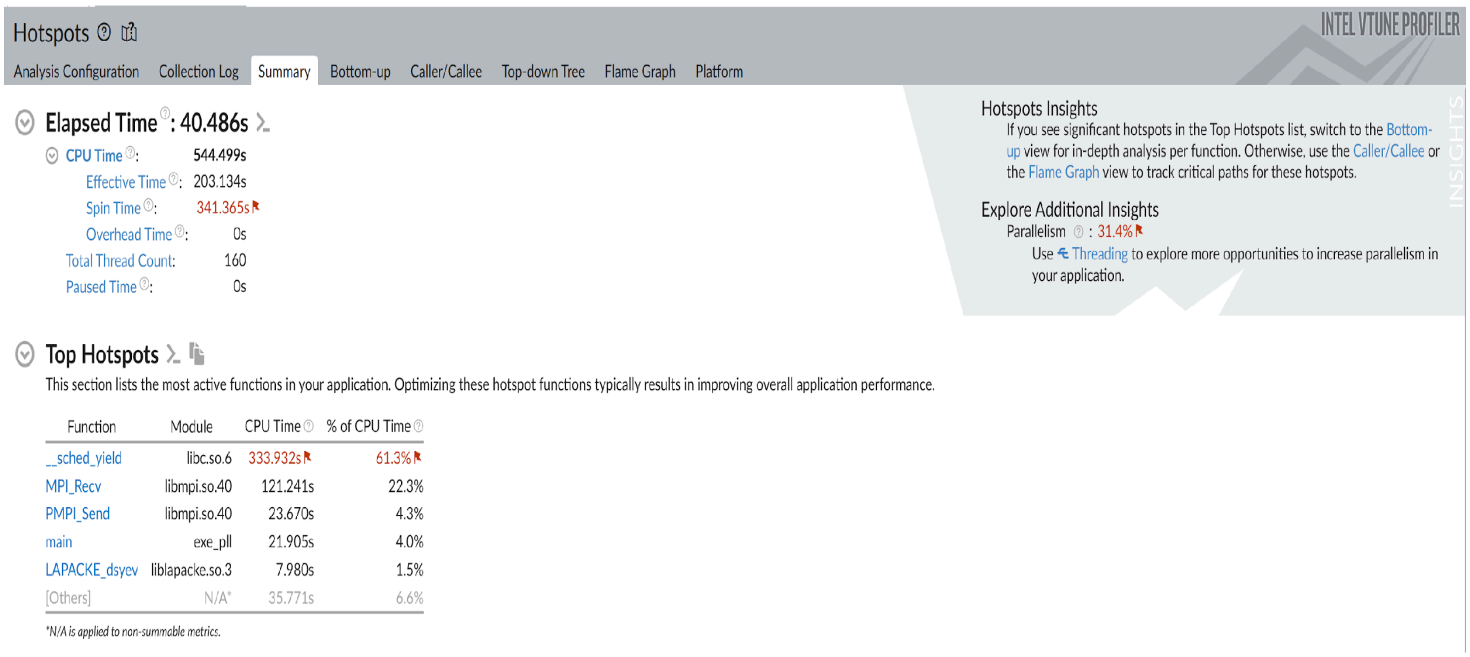
Name:	Intel(R) microarchitecture code named Tigerlake H
Frequency:	2.3 GHz
Logical CPU Count:	16

Cache Allocation Technology

Level 2 capability:	available
Level 3 capability:	not detected

4.5.2 MPI parallel code profiling

Software used: Intel's Vtune Profiler



Hotspots

INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

Elapsed Time: 40.486s

CPU Time:	544.499s
Effective Time:	203.134s
Spin Time:	341.365s
Overhead Time:	0s
Total Thread Count:	160
Paused Time:	0s

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism: 31.4%

Use Threading to explore more opportunities to increase parallelism in your application.

Top Hotspots

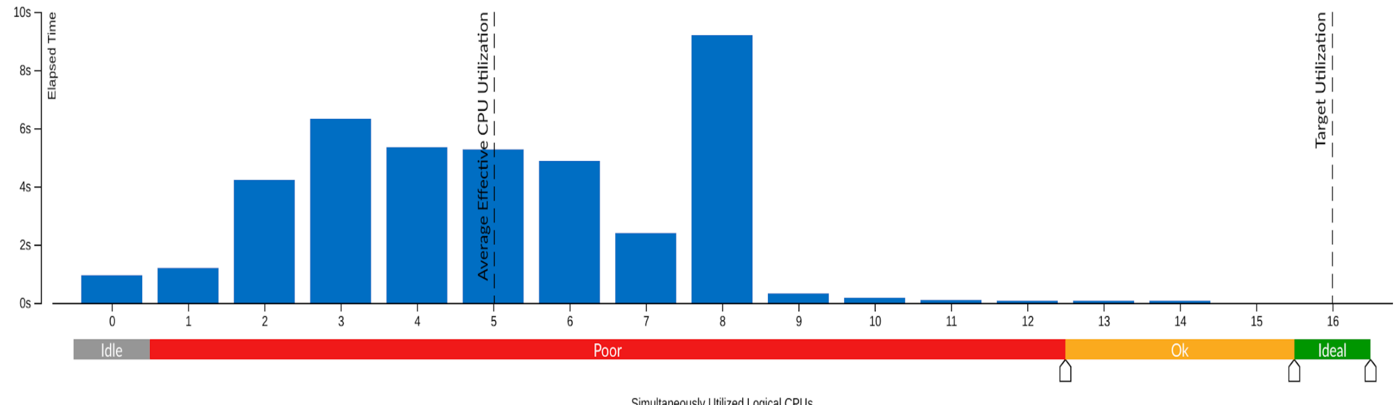
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
__sched_yield	libc.so.6	333.932s	61.3%
MPI_Recv	libmpi.so.40	121.241s	22.3%
PMPI_Send	libmpi.so.40	23.670s	4.3%
main	exe_pll	21.905s	4.0%
LAPACKE_dsyev	liblapacke.so.3	7.980s	1.5%
[Others]	N/A*	35.771s	6.6%

*N/A is applied to non-summable metrics.

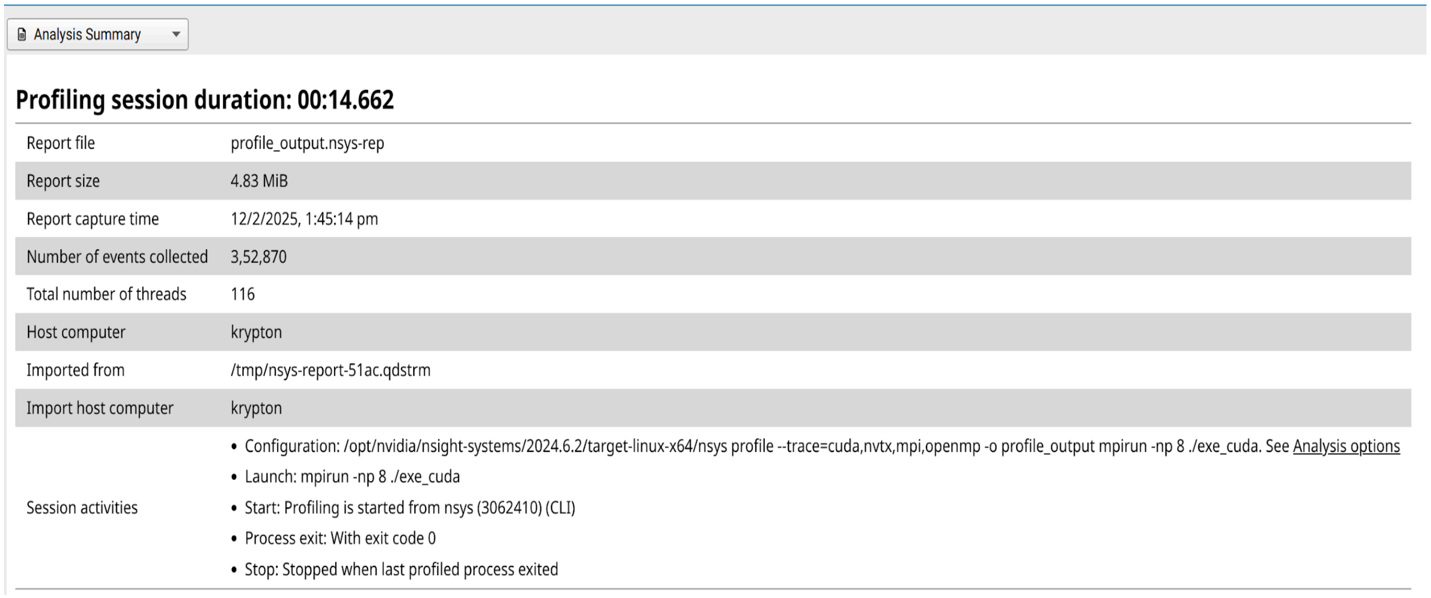
Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



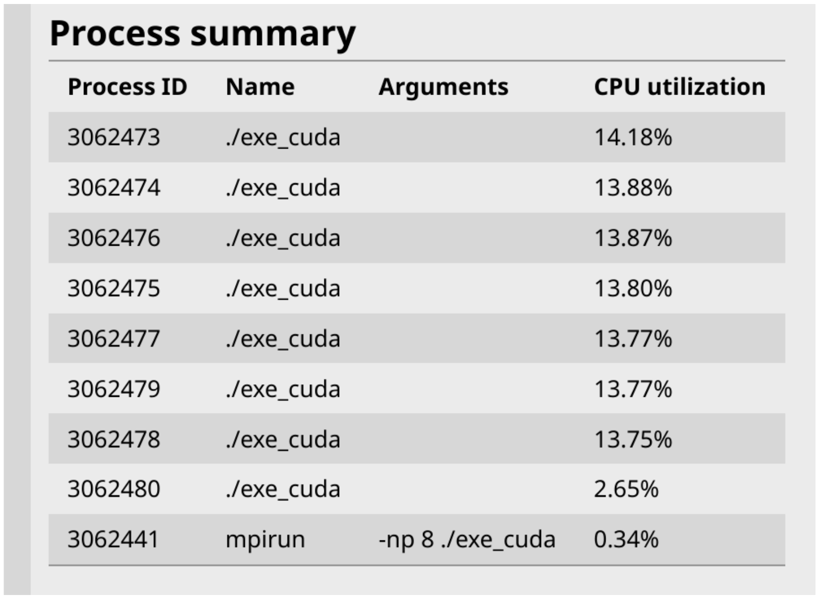
4.5.3 MPI + CUDA parallel code profiling

Software used: Nvidia's Nsight Systems



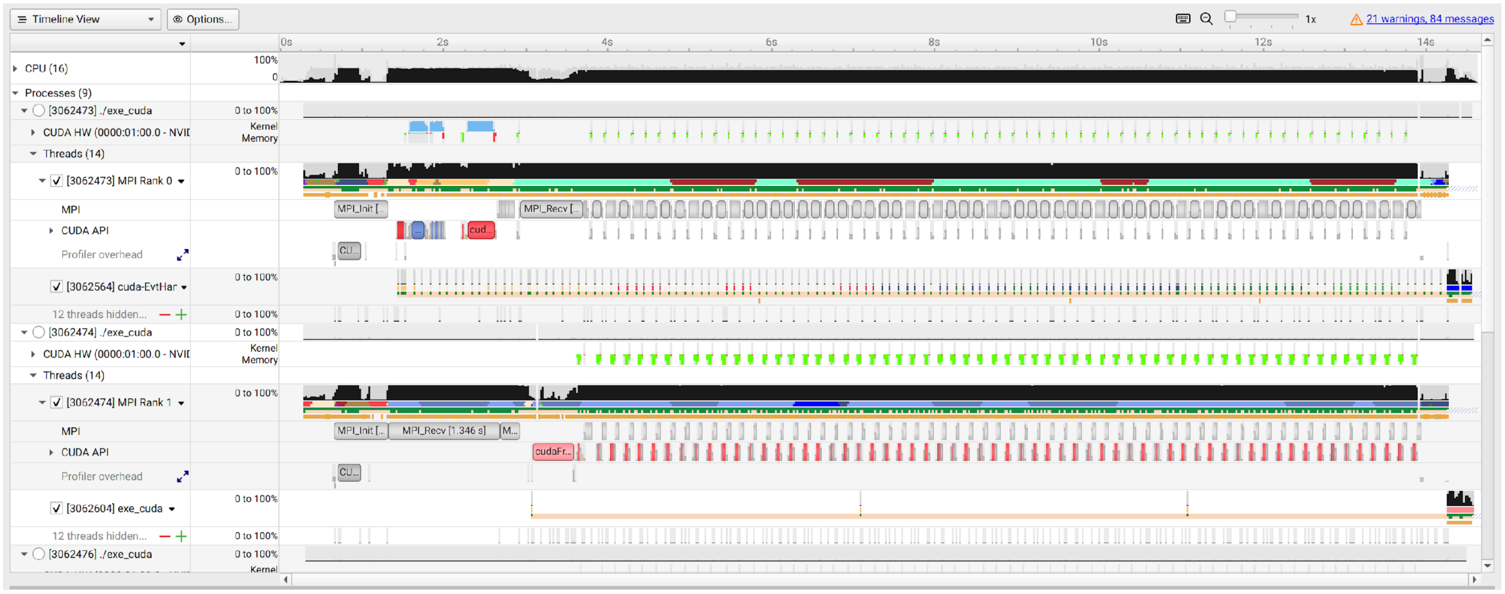
Profiling session duration: 00:14.662

Report file	profile_output.nsys-rep
Report size	4.83 MiB
Report capture time	12/2/2025, 1:45:14 pm
Number of events collected	3,52,870
Total number of threads	116
Host computer	krypton
Imported from	/tmp/nsys-report-51ac.qdstrm
Import host computer	krypton
Session activities	• Configuration: /opt/nvidia/nsight-systems/2024.6.2/target-linux-x64/nsys profile --trace=cuda,nvtx,mpi,openmp -o profile_output mpirun -np 8 ./exe_cuda. See Analysis options • Launch: mpirun -np 8 ./exe_cuda • Start: Profiling is started from nsys (3062410) (CLI) • Process exit: With exit code 0 • Stop: Stopped when last profiled process exited



Process summary

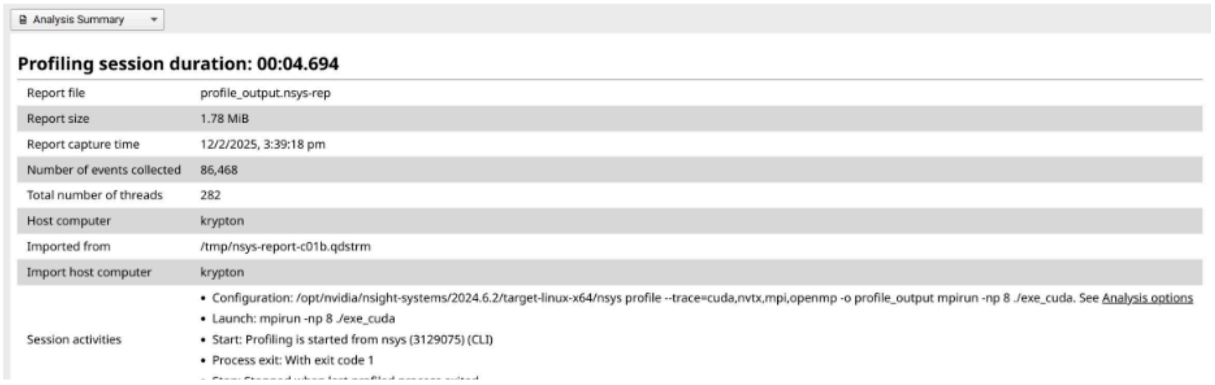
Process ID	Name	Arguments	CPU utilization
3062473	./exe_cuda		14.18%
3062474	./exe_cuda		13.88%
3062476	./exe_cuda		13.87%
3062475	./exe_cuda		13.80%
3062477	./exe_cuda		13.77%
3062479	./exe_cuda		13.77%
3062478	./exe_cuda		13.75%
3062480	./exe_cuda		2.65%
3062441	mpirun	-np 8 ./exe_cuda	0.34%



Timeline View
Options...
1x
21 warnings, 84 messages
0s
2s
4s
6s
8s
10s
12s
14s
CPU (16)
100%
0
Processes (9)
[3062473] ./exe_cuda
0 to 100%
CUDA HW (0000:01:00.0 - NVIDI
Kernel
Memory
Threads (14)
[3062473] MPI Rank 0
MPI
MPI_Init [...
MPI_Recv [...
CUDA API
cud...
Profiler overhead
CU...
[3062564] cuda-EvtHar
12 threads hidden...
[3062474] ./exe_cuda
[3062474] MPI Rank 1
MPI_Recv [1.346 s]
M...
cudaFr...
[3062604] exe_cuda
[3062476] ./exe_cuda

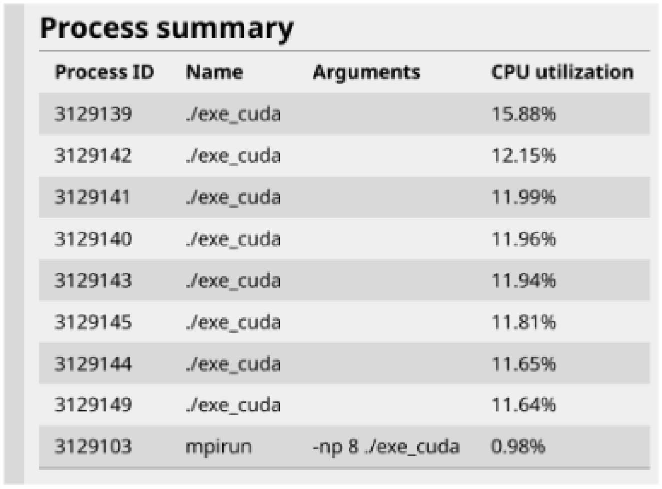
4.5.3 MPI + CUDA + openMP parallel code profiling

Software used: Nvidia’s Nsight Systems



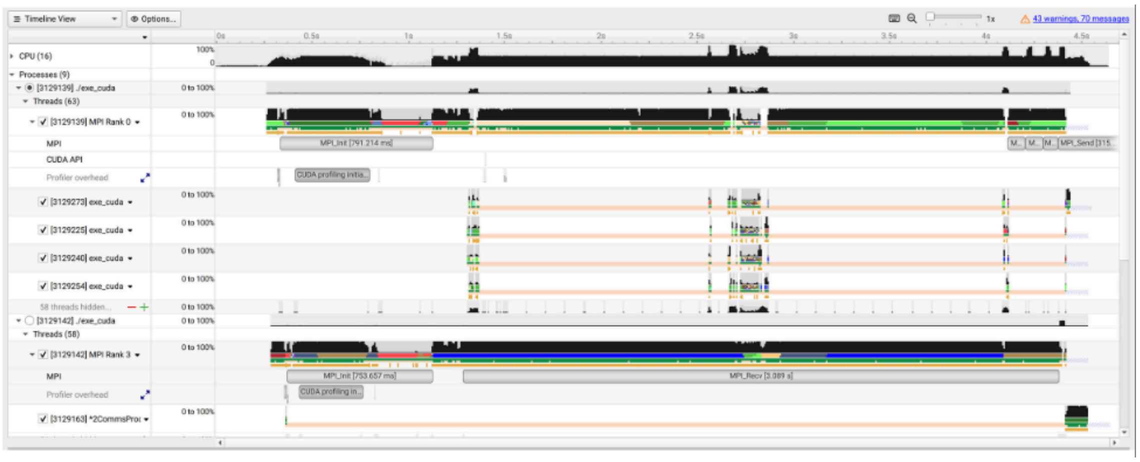
Profiling session duration: 00:04.694

Report file	profile_output.nsys-rep
Report size	1.78 MiB
Report capture time	12/2/2025, 3:39:18 pm
Number of events collected	86,468
Total number of threads	282
Host computer	krypton
Imported from	/tmp/nsys-report-c01b.qdstrm
Import host computer	krypton
Session activities	• Configuration: /opt/nvidia/nsight-systems/2024.6.2/target-linux-x64/nsys profile --trace=cuda,nvtx,mpi,openmp -o profile_output mpirun -np 8 ./exe_cuda. See Analysis options • Launch: mpirun -np 8 ./exe_cuda • Start: Profiling is started from nsys (3129075) (CLI) • Process exit: With exit code 1



Process summary

Process ID	Name	Arguments	CPU utilization
3129139	./exe_cuda		15.88%
3129142	./exe_cuda		12.15%
3129141	./exe_cuda		11.99%
3129140	./exe_cuda		11.96%
3129143	./exe_cuda		11.94%
3129145	./exe_cuda		11.81%
3129144	./exe_cuda		11.65%
3129149	./exe_cuda		11.64%
3129103	mpirun	-np 8 ./exe_cuda	0.98%



Chapter 5

Conclusions

5.1 Conclusions

In this study, we systematically evaluated the execution time of three different implementations of our scientific simulation program:

- Sequential Implementation: A baseline version that runs on a single CPU core without any parallelization.
- MPI-based Parallel Implementation: A parallel version using MPI, which enables computation across multiple processes.
- Hybrid CUDA + MPI + openMP Implementation: A combination of GPU acceleration (CUDA) and MPI + openMP CPU parallelization to optimize computational efficiency.

The most computationally expensive step in the simulation was the matrix diagonalization operation, which is a crucial component in solving large-scale scientific problems. This step was significantly optimized by using CUDA's cuSolver function, resulting in a 36x speedup compared to the sequential version.

The performance benefit of MPI-based parallelization became more significant as the matrix size increased. While for smaller matrices, the overhead of inter-process communication limited speedup, for matrices larger than 6000×6000, MPI provided a noticeable reduction in execution time.

The combined approach using CUDA, MPI, and OpenMP delivered the most optimized performance. For a matrix size of 6500×6500, the hybrid approach achieved a 2.8× overall speedup compared to the sequential version. This was due to an efficient division of computational workload:

- CUDA handled matrix operations on GPUs, significantly accelerating linear algebra computations.
- MPI ensured efficient parallel execution across multiple CPU cores, reducing execution time for large problem sizes.
- OpenMP optimized CPU parallelization utilizing multi-threading to reduce execution bottlenecks.

This study demonstrates that a hybrid approach combining CUDA, MPI, and OpenMP offers the best performance for large-scale matrix computations. By utilizing heterogeneous computing resources (CPU + GPU), we can achieve a highly scalable and efficient solution, making it suitable for computationally intensive scientific simulations.

References

[1] Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.
ISBN-10: 0123742609, ISBN-13 : 978-0123742605

[2] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., & McDonald, J. (2001). *Parallel Programming in OpenMP*. Morgan Kaufmann.
ISBN-10: 1558606718, ISBN-13: 978-1558606715

[3] Sanders, J. & Kandrot, E. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley.
ISBN-10: 0131387685, ISBN-13: 978-0131387683