High Performance Computing

CUDA Practice: Parallelization of Heat Diffussion Equation

Ariza Pérez, José Ramon
Buturuga, Mihaela Alexandra

University of Lleida
Higher Polytechnic School
Master's Degree in Informatics Engineering

Course 2024 - 2025

June 8th, 2025

Contents

1 Serial Code

This C program simulates heat diffusion in a 2D square area using a grid. It starts with two hot diagonal lines on a cold background. The program then calculates how the temperature changes over time, updating each point based on its neighbors, while keeping the edges cold. Finally, it creates a color image (BMP) showing the final heat distribution, with red being hot and blue being cold.

2 Analysis of the Hotspots

In the analysis of the heat diffusion simulation implemented in C, the main hotspots identified for parallelization using CUDA are the initialization of the matrix and the solving of the heat equation. These sections are the most computationally intensive, especially with large grid sizes due to the repetitive operations over each cell.

By offloading these computations to the GPU using CUDA, we aim to significantly reduce runtime by exploiting the massive parallelism of modern GPUs. CUDA allows us to perform thousands of updates concurrently by launching kernels that process the grid in parallel, thus accelerating the simulation and improving scalability on hardware with high-throughput capabilities.

2.1 Initialization of the Heat Matrix

The initialization of the matrix remains an ideal candidate for parallelization in the CUDA implementation. This phase consists of assigning initial temperature values to each cell in the grid prior to the simulation. Since each cell is initialized independently, the task is embarrassingly parallel, with no data dependencies involved.

In this CUDA-based approach, the entire grid is allocated in GPU global memory and a CUDA kernel is launched to initialize the matrix in parallel. Each thread is assigned to compute the initial value of a single cell based on its coordinates. This strategy leverages the massive thread parallelism of the GPU to significantly reduce the setup time of the simulation and to ensure efficient utilization of GPU resources from the very beginning of the execution.

2.2 Solving the Heat Diffusion Equation

The heat diffusion equation is solved iteratively by updating the temperature of each cell based on the values of its immediate neighbors. In the GPU-based implementation, this process is fully parallelized: each thread is responsible for updating a single grid cell, provided it is not located at the boundary.

Because each update depends only on the values from the previous time step, the computation is inherently parallel and well-suited to execution on the GPU. Boundary cells are excluded from the update phase and are treated separately to enforce the fixed boundary conditions.

After each iteration, the updated values are written to a separate grid to avoid overwriting data needed by neighboring threads. A pointer swap between the current and new grids is performed at the end of each step to prepare for the next iteration. This approach ensures data consistency and leverages the GPU's parallel architecture to accelerate the simulation.

2.3 Transforming the Heat Matrix into a BMP File

The final step of the simulation involves converting the temperature matrix into a BMP image. This step is intentionally not parallelized in any of the implementations, including the CUDA version. While minor speedups could be achieved by parallelizing this stage, the added complexity and minimal performance gain did not justify the change.

In the OpenMP and hybrid MPI+OpenMP version, and now in the CUDA implementation, the BMP file is generated sequentially by the main process. This approach ensures consistency across all tested versions and allows for a fair comparison of computation times, since none of them include GPU or multi-threaded I/O acceleration.

Additionally, avoiding parallelization in this stage helps clearly separate the computational workload, where acceleration is meaningful, from I/O operations, which are inherently more sequential and less significant in terms of total runtime. It also avoids potential issues with concurrent file access or complex coordination between processes or threads.

3 Parallelization

3.1 Function `initialize_grid`

The `initialize_grid` function (Code 1) assigns the initial temperature values to the entire simulation grid in parallel using CUDA. It is implemented as a kernel in which each thread is responsible for initializing a single cell of the 2D matrix.

The grid is stored in global memory, as it needs to be accessible across the entire kernel and reused in subsequent computations. Each thread calculates its unique 2D position `(i, j)` based on its block and thread indices.

The initialization logic assigns a constant temperature T to cells located on either of the two main diagonals of the matrix, and zero to all others. Since each cell's value is independent of the others, the kernel requires no synchronization, allowing maximum parallelism.

3.2 Function `solve_heat_equation` (Implemented in `main`)

The heat equation is solved iteratively within the `main` function (Code 2) using two CUDA kernels: `solve_step` (Code 3) and `apply_boundary_conditions` (Code 4). Each iteration of the simulation consists of updating the interior points of the grid based on the diffusion formula and then applying Dirichlet boundary conditions.

The `solve_step` kernel performs the core computation of the heat diffusion equation. It uses 2D CUDA thread blocks to assign a thread to each grid point in the domain. Each thread computes the updated temperature at a specific cell using the values of its four direct neighbors from the previous state. The kernel reads from the current grid and writes the result to a separate output grid. To avoid invalid memory access, the kernel only updates interior points, skipping the boundary cells.

After the update step, the `apply_boundary_conditions` kernel sets the values on the boundaries of the grid to zero, ensuring fixed Dirichlet conditions. A 1D kernel is used for simplicity, where each thread handles a cell on the top/bottom rows or left/right columns, depending on its index. Since each thread writes to a unique memory location, there is no need for synchronization.

A pointer swap is performed after each iteration to alternate the roles of the current and next grids, avoiding unnecessary memory copies. This approach efficiently exploits the parallelism of CUDA while keeping the logic straightforward and the memory usage minimal.

4 Performance

To evaluate the performance of the CUDA implementation of the heat diffusion simulation, different thread block configurations were tested. The objective was to analyze how the GPU scales with increasing parallelism and to identify optimal configurations for distributing the workload.

A set of 2D block sizes was chosen to progressively double the number of threads per kernel launch, starting from 2 up to 1024 total threads:

Block Size (X, Y)	**Threads**	**Block Size (X, Y)**	**Threads**
(2, 1)	2	(8, 8)	64
(2, 2)	4	(16, 8)	128
(4, 2)	8	(16, 16)	256
(4, 4)	16	(32, 16)	512
(8, 4)	32	(32, 32)	1024

Table 1: CUDA Thread Block Configurations

These configurations were selected to gradually increase the parallel workload and to observe how performance changes as the GPU is more fully utilized.

The chosen shapes maintain a relatively balanced aspect ratio (close to square or rectangular) to improve memory coalescing and reduce warp divergence, which can negatively impact performance. Each configuration also respects the maximum number of threads per block imposed by CUDA (typically 1024).

This approach leverages CUDA's architecture, where many blocks can be distributed simultaneously across the Streaming Multiprocessors (SMs). It allows the GPU to overlap memory operations and computations efficiently and helps achieve higher throughput as the number of threads increases.

4.1 Execution Time

Execution time increases proportionally with both matrix size and number of steps, as shown in Table 2. A tenfold increase in steps leads to roughly ten times longer runtimes, while increasing matrix dimensions (e.g., from 100×100 to 1000×1000) results in a near-proportional rise due to the growth in total grid cells.

This highlights the computational intensity of large-scale simulations and the need for parallelization. CUDA results (Table 3, Table 4, Table 5) confirm that GPU acceleration drastically reduces execution time compared to the serial approach, particularly for large matrices and high step counts.

Some key observations about the CUDA results are:

- **Significant speedup over serial code:** CUDA execution times are orders of magnitude faster than serial runs, especially for large matrices and high step counts. For example, for the 2000×2000 grid with 100,000 steps, the CUDA kernel with 32 threads per block executes in about 33.11 seconds, drastically reducing runtime compared to serial execution.

- **Performance gains saturate beyond moderate thread counts:** Increasing the total number of CUDA threads beyond a certain point (e.g., from 256 to 1024 threads per block) yields diminishing returns and sometimes slight performance degradation due to increased scheduling overhead and resource contention within the GPU.

- **Smaller problems show less pronounced improvements:** For small grid sizes such as 100×100, CUDA achieves very low execution times overall, but performance differences across thread configurations are minimal, reflecting the limited computational workload and higher relative overhead.

- **Optimal thread configuration varies with problem size:** Larger problems benefit more noticeably from increased thread parallelism up to a threshold. For example, in the 1000×1000 and 2000×2000 grids, the best performance occurs around 32 to 64 threads per block, balancing workload distribution and GPU resource utilization efficiently.

- **Scalability with step count:** Execution time scales roughly linearly with the number of simulation steps, consistent with the computational workload model, confirming the CUDA kernels' efficient utilization of GPU parallelism even for large iteration counts.

- **Low overhead and good resource utilization:** CUDA leverages fine-grained parallelism with thousands of lightweight threads and fast memory access patterns, resulting in consistently low execution times and good scaling compared to CPU parallel implementations.

- **Compared to OpenMP and hybrid MPI+OpenMP:** CUDA drastically reduces execution time across all problem sizes. While OpenMP and hybrid approach offered improvements over serial execution, CUDA provides an order-of-magnitude reduction in runtime, especially for large matrices and high step counts, thanks to its massive parallelism and specialized hardware.

  Some examples of these lower results compared to CUDA can be shown in Table 6 and Table 7, where the best obtained results of the OpenMP and hybrid OpenMP + MPI for the biggest problem size are located, respectively.

4.2 Speedup and Scalability

To evaluate parallel performance, we measure speedup, defined as $\text{Speedup}(p) = T_{\text{serial}}/T_p$, where T_{serial} is the execution time of the serial version and T_p is the time using p CUDA threads.

Speedup results are presented in Table 8, Table 9 and Table 10, with corresponding charts in Figure 1, Figure 2 and Figure 3.

The analysis of the CUDA speedup results reveals several important patterns:

- **Impact of matrix size:** Speedup improves significantly with increasing matrix size. For the smallest matrix (100×100), speedups are mostly below or around 1 for low step counts, indicating that parallel overhead dominates computation. For intermediate (1000×1000) and large (2000×2000) matrices, speedups increase substantially, especially at higher step counts, showing that larger workloads better amortize GPU overhead.

- **Effect of step count:** Increasing the number of steps generally improves speedup. For example, in the 2000×2000 matrix, speedup grows from around 11 at 10,000 steps to over 130 at 100,000 steps (with 64 threads), showing strong scalability with workload. However, for very large step counts, speedup may slightly decrease due to memory bandwidth saturation or GPU resource contention.

- **Optimal number of threads:** Configurations with 32 to 64 CUDA threads often achieve the best performance across matrix sizes. Higher thread counts (128 to 1024) show diminishing returns or even decreased speedup, likely due to increased contention and scheduling overhead on the GPU.

- **Performance degradation for small matrices:** For the 100×100 matrix, speedup remains below 1 for low step counts (e.g., only 0.14–0.16 at 100 steps), reflecting that kernel launch overhead and memory setup costs dominate in small-scale computations.

- **Scalability for large matrices:** For the 2000×2000 matrix, speedup increases almost linearly with the number of steps for mid-range thread configurations. For example, speedup grows from around 11 (10,000 steps) to over 130 (100,000 steps) with 64 threads, demonstrating excellent scalability of the CUDA implementation.

- **Memory and scheduling effects:** The slight reduction in speedup at very high thread counts (e.g., 1024 threads) and step counts may be attributed to the GPU's hardware limits on active threads per block and shared memory usage. This emphasizes the need to balance parallelism and hardware constraints for optimal performance.

- **Compared to OpenMP and hybrid MPI+OpenMP:** CUDA achieves significantly higher speedup, especially on large matrices (e.g., 1000×1000 and 2000×2000). Its massive parallelism allows it to outperform CPU-based models by a substantial margin.

  Some examples of these lower results compared to CUDA can be shown in Figure 4 and Figure 5, where the best obtained results of the OpenMP and hybrid OpenMP + MPI for the biggest problem size are located, respectively.

In summary, CUDA demonstrates excellent scalability, particularly for large matrices and high computational workloads, validating its effectiveness for accelerating stencil-based simulations.

4.3 Efficiency

Parallel efficiency, $E_p = S_p/p$, measures how effectively processing resources are used compared to ideal linear scaling ($E_p = 1$). Efficiency results are shown in Table 11, Table 12 and Table 13, with corresponding charts in Figure 6, Figure 7 and Figure 8.

Analyzing the CUDA-based parallel efficiency we observe the following:

- **Efficiency rapidly deteriorates at high thread counts:** For all matrix sizes, increasing the number of CUDA threads beyond a small number leads to sharp efficiency drops. This is most evident in small matrices, where overhead dominates due to low per-thread workload, but also appears in larger problems as thread counts grow excessively.

- **Larger problem sizes better utilize GPU parallelism:** As the matrix size increases, the GPU has more meaningful work per thread, allowing overhead to be amortized. For example, at 100,000 steps, the 2000×2000 matrix achieves significantly higher efficiency (above 5.0 with a few threads), compared to under 3.5 for the 100×100 matrix.

- **There is a sweet spot in step count for each problem size:** Increasing the number of steps improves efficiency up to a point by shifting the runtime balance from overhead to computation. However, the marginal benefit plateaus, especially in small matrices, where even long simulations cannot saturate the GPU efficiently.

- **Efficiency scaling is sublinear and sometimes regressive:** Adding more threads does not guarantee better speedup per resource. In fact, for large matrices at high step counts, efficiency peaks with just a few threads and then steadily declines, revealing the non-linear and even regressive scaling behavior of CUDA when resource contention and memory bottlenecks arise.

- **Hardware saturation limits efficiency in small problems:** The smallest problem (100×100) consistently fails to reach high efficiency, regardless of simulation length or thread count. This suggests a structural mismatch between problem granularity and GPU architecture: the available threads are underutilized due to insufficient computational demand.

- **Optimal performance often comes from conservative parallelism:** Surprisingly, the best efficiencies are not achieved by maximizing thread count, but rather by using a modest number of threads on sufficiently large workloads. This emphasizes that aggressive parallelism can hurt performance when it overwhelms memory or scheduling resources.

- **Compared to OpenMP and hybrid MPI+OpenMP:** CUDA shows higher efficiency for compute-intensive workloads, reaching better utilization of resources than OpenMP or hybrid models. Despite requiring careful tuning, CUDA delivers superior performance and efficiency on large-scale problems.

Some examples of these lower results compared to CUDA can be shown in Figure 9 and Figure 10, where the best obtained results of the OpenMP and hybrid OpenMP + MPI for the biggest problem size are located, respectively.

In summary, CUDA efficiency is highest when matrix size, thread count and step count are well-balanced. Over-parallelization, especially in small problems, leads to poor performance due to overhead and limited workload per thread.

5 Conclusions

In this project, we compared three parallel computing approaches (OpenMP, OpenMP + MPI and CUDA) to solve the heat equation problem, which is a numerical problem. The results clearly demonstrate that CUDA provides the best performance by far.

Execution times were significantly lower with CUDA and both speedup and efficiency were markedly higher compared to CPU-based approaches, especially for large matrices and long simulations. This highlights the immense computational power of GPUs when the workload is well-suited for massive parallelism.

OpenMP proved to be a simple and quick solution for parallelizing on multicore CPUs, showing reasonable speedups for small to medium problem sizes. However, its scalability was limited due to shared-memory constraints. The hybrid OpenMP+MPI model extended scalability across nodes, allowing larger problem sizes to be handled more effectively. Still, it introduced more complexity and communication overhead, which limited its efficiency gains compared to CUDA.

From the programmer's point of view, GPU development using CUDA requires an understanding of specific concepts like memory management, kernel design, and thread organization. Despite this, adapting our code to CUDA was not especially difficult. With a clear structure in place, the translation to GPU kernels was relatively straightforward. Given the outstanding performance improvements observed in this delivery, the cost-benefit tradeoff is very favorable: the effort required to adapt the code is minimal compared to the performance gains achieved.

In conclusion, while CPU-based models like OpenMP and MPI are easier to implement and still valuable for many applications, using GPUs through CUDA is clearly worth it for compute-intensive workloads. The results of this project confirm that investing in GPU programming can lead to substantial benefits in both execution time and scalability.

Appendix

A Functions

A.1 Function `initialize_grid`

```
__global__ void initialize_grid(double *grid, int nx, int ny) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < nx && j < ny) {
        if (i == j || i == nx - 1 - j)
            grid[i * ny + j] = T;
        else
            grid[i * ny + j] = 0.0;
    }
}
```

Code 1: Function `initialize_grid` parallelized with CUDA

A.2 Function `solve_heat_equation` (Implemented in `main`)

```
int main(int argc, char *argv[]) {
    ...
    // Solve heat equation
    for (int step = 0; step < steps; step++) {
        solve_step<<<numBlocks, threadsPerBlock>>>(d_grid, d_new_grid, nx, ny, r
            );
        apply_boundary_conditions<<<numBlocks, threadsPerBlock>>>(d_new_grid, nx
            , ny);
        double *temp = d_grid;
        d_grid = d_new_grid;
        d_new_grid = temp;
    }
    ...
}
```

Code 2: Function `solve_heat_equation` (Implemented in `main`) parallelized with CUDA

A.3 Function `solve_step`

```
__global__ void solve_step(double *grid, double *new_grid, int nx, int ny,
    double r) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (i > 0 && i < nx - 1 && j > 0 && j < ny - 1) {
        new_grid[i * ny + j] = grid[i * ny + j]
            + r * (grid[(i + 1) * ny + j] + grid[(i - 1) * ny + j] - 2 * grid[i
                * ny + j])
            + r * (grid[i * ny + j + 1] + grid[i * ny + j - 1] - 2 * grid[i * ny
                 + j]);
    }
}
```

Code 3: Function `solve_step` parallelized with CUDA

A.4 Function `apply_boundary_conditions`

```
__global__ void apply_boundary_conditions(double *grid, int nx, int ny) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < nx) {
        grid[0 * ny + idx] = 0.0;
        grid[(nx - 1) * ny + idx] = 0.0;
    }
    if (idx < ny) {
        grid[idx * ny + 0] = 0.0;
        grid[idx * ny + (ny - 1)] = 0.0;
    }
}
```

Code 4: Function `apply_boundary_conditions` parallelized with CUDA

B Tables

B.1 Execution Times

B.1.1 Serial Execution Time

	Steps			
Matrix Size	**100**	**1,000**	**10,000**	**100,000**
100 x 100	0.01	0.11	1.02	10.20
1000 x 1000	1.17	12.45	110.92	1,073.50
2000 x 2000	4.66	49.19	476.57	4,314.34

Table 2: Serial Execution Time (in seconds)

B.1.2 Execution Time (CUDA) in a 100x100 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	0.07	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
1,000	0.08	0.08	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07
10,000	0.23	0.15	0.12	0.12	0.11	0.11	0.12	0.11	0.14	0.16
100,000	1.46	0.90	0.63	0.55	0.55	0.54	0.54	0.53	0.69	0.80

Table 3: CUDA Execution Time (in seconds) for 100×100 matrix

B.1.3 Execution Time (CUDA) in a 1000x1000 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	0.25	0.20	0.18	0.16	0.15	0.15	0.16	0.16	0.17	0.17
1,000	1.23	0.68	0.41	0.29	0.22	0.23	0.25	0.27	0.29	0.30
10,000	10.64	5.45	2.82	1.55	0.94	0.93	1.18	1.13	1.69	1.76
100,000	106.88	54.01	27.62	14.64	8.51	8.31	10.81	10.35	15.96	16.66

Table 4: CUDA Execution Time (in seconds) for 1000×1000 matrix

B.1.4 Execution Time (CUDA) in a 2000x2000 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	0.81	0.60	0.49	0.46	0.42	0.42	0.43	0.43	0.45	0.46
1,000	4.64	2.50	1.44	0.92	0.69	0.68	0.78	0.77	0.97	1.00
10,000	42.70	21.74	11.23	6.04	3.61	3.58	4.52	4.42	6.44	6.72
100,000	427.66	214.60	109.77	57.40	33.11	32.56	42.00	41.09	61.43	64.12

Table 5: CUDA Execution Time (in seconds) for 2000×2000 matrix

B.1.5 Execution Time (OpenMP) in a 2000x2000 Matrix

	Threads		
Steps	**2**	**4**	**8**
100	3.26	1.93	1.99
1,000	31.02	16.09	16.74
10,000	306.80	156.84	166.32
100,000	2,834.26	1,480.20	1,523.59

Table 6: Parallel Execution Time (in seconds) with OpenMP for 2000x2000 matrix

B.1.6 Execution Time (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

	MPI Tasks (2 OpenMP Threads)				
Steps	**2**	**4**	**8**	**16**	**32**
100	4.60	2.56	2.35	2.06	2.47
1,000	42.05	21.76	17.90	14.22	12.82
10,000	415.33	233.92	176.83	138.48	117.71
100,000	4,052.52	2,555.92	1,826.81	1,325.09	1,143.02

Table 7: Execution Time with OpenMP (2 threads) + MPI Tasks for 2000×2000 matrix

B.2 Speedup

B.2.1 Speedup (CUDA) in a 100x100 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	0.14	0.15	0.16	0.16	0.15	0.16	0.15	0.16	0.15	0.15
1,000	1.37	1.43	1.55	1.53	1.58	1.59	1.51	1.60	1.54	1.52
10,000	4.46	6.95	8.50	8.75	9.07	9.13	8.69	9.06	7.05	6.43
100,000	6.97	11.33	16.14	18.67	18.60	18.87	18.97	19.15	14.72	12.72

Table 8: CUDA Speedup for 100×100 matrix

B.2.2 Speedup (CUDA) in a 1000x1000 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	4.71	5.89	6.60	7.10	7.60	7.64	7.30	7.20	7.02	7.02
1,000	10.09	18.30	30.04	43.23	56.28	53.30	49.47	46.28	42.33	41.08
10,000	10.43	20.37	39.31	71.74	117.70	119.28	93.61	97.74	65.57	63.04
100,000	10.04	19.88	38.87	73.31	126.13	129.12	99.29	103.76	67.27	64.45

Table 9: CUDA Speedup for 1000×1000 matrix

B.2.3 Speedup (CUDA) in a 2000x2000 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	5.73	7.80	9.50	10.04	11.10	11.08	10.84	10.83	10.28	10.15
1,000	10.60	19.68	34.07	53.30	71.48	72.23	63.30	64.04	50.87	49.25
10,000	11.16	21.92	42.44	78.96	131.83	133.27	105.53	107.74	73.98	70.94
100,000	10.09	20.10	39.30	75.17	130.31	132.52	102.73	104.99	70.23	67.28

Table 10: CUDA Speedup for 2000×2000 matrix

B.3 Efficiency

B.3.1 Efficiency (CUDA) in a 100x100 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	0.07	0.04	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00
1,000	0.69	0.36	0.19	0.10	0.05	0.02	0.01	0.01	0.00	0.00
10,000	2.23	1.74	1.06	0.55	0.28	0.14	0.07	0.04	0.01	0.01
100,000	3.49	2.83	2.02	1.17	0.58	0.29	0.15	0.07	0.03	0.01

Table 11: CUDA Efficiency for 100×100 matrix

B.3.2 Efficiency (CUDA) in a 1000x1000 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	2.35	1.47	0.83	0.44	0.24	0.12	0.06	0.03	0.01	0.01
1,000	5.04	4.57	3.76	2.70	1.76	0.83	0.39	0.18	0.08	0.04
10,000	5.21	5.09	4.91	4.48	3.68	1.86	0.73	0.38	0.13	0.06
100,000	5.02	4.97	4.86	4.58	3.94	2.02	0.78	0.41	0.13	0.06

Table 12: CUDA Efficiency for 1000×1000 matrix

B.3.3 Efficiency (CUDA) in a 2000x2000 Matrix

	CUDA Threads									
Steps	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
100	2.86	1.95	1.19	0.63	0.35	0.17	0.08	0.04	0.02	0.01
1,000	5.30	4.92	4.26	3.33	2.23	1.13	0.49	0.25	0.10	0.05
10,000	5.58	5.48	5.30	4.93	4.12	2.08	0.82	0.42	0.14	0.07
100,000	5.04	5.03	4.91	4.70	4.07	2.07	0.80	0.41	0.14	0.07

Table 13: CUDA Efficiency for 2000×2000 matrix

C Charts

C.1 Speedup

C.1.1 Speedup (CUDA) in a 100x100 Matrix

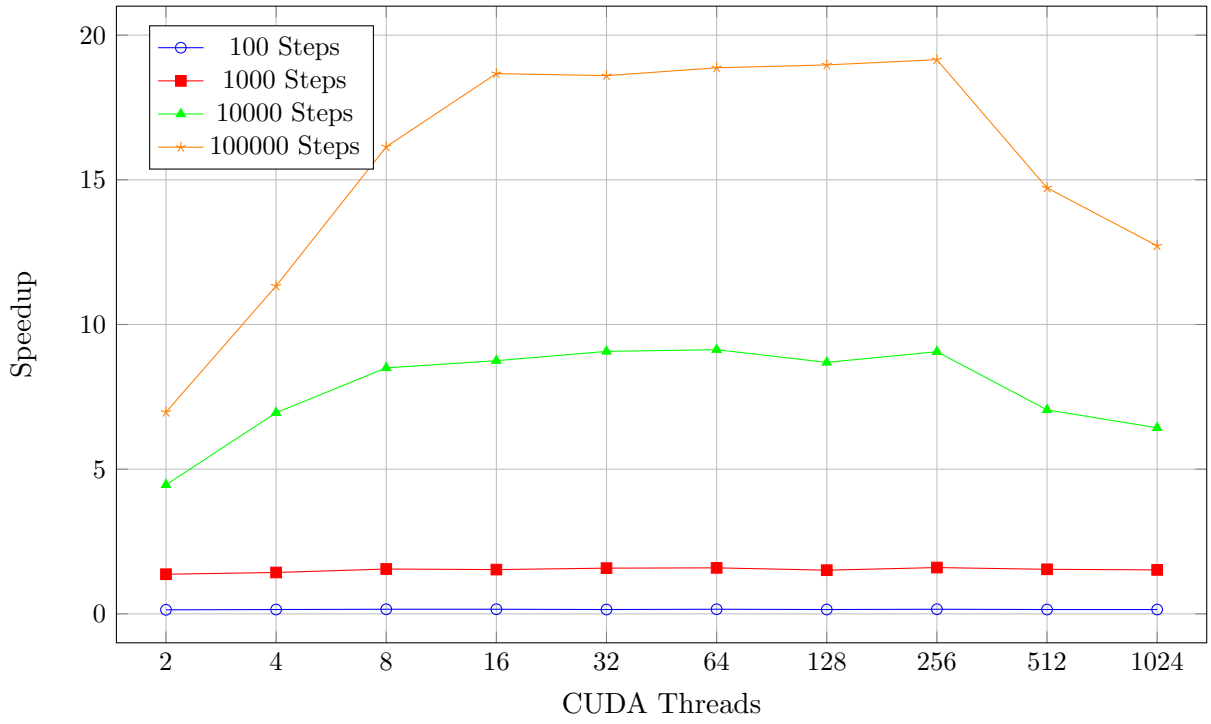

Figure 1: CUDA Speedup for 100×100 matrix

C.1.2 Speedup (CUDA) in a 1000x1000 Matrix

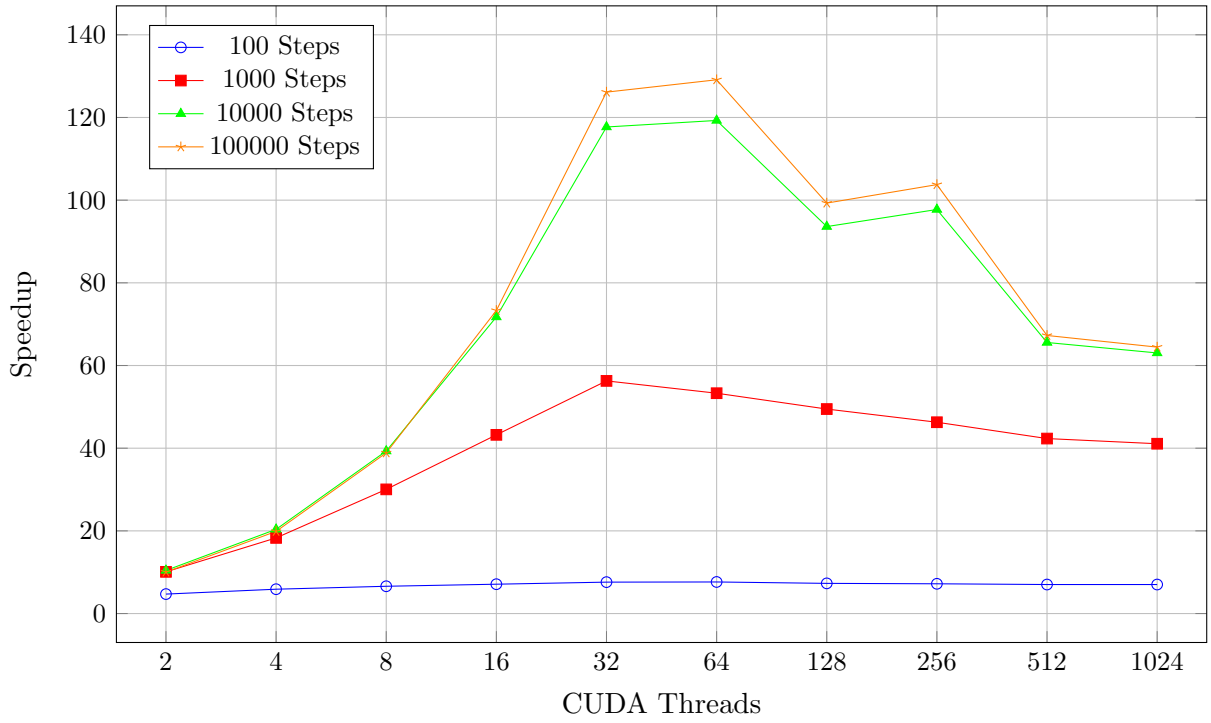

Figure 2: CUDA Speedup for 1000×1000 matrix

C.1.3 Speedup (CUDA) in a 2000x2000 Matrix

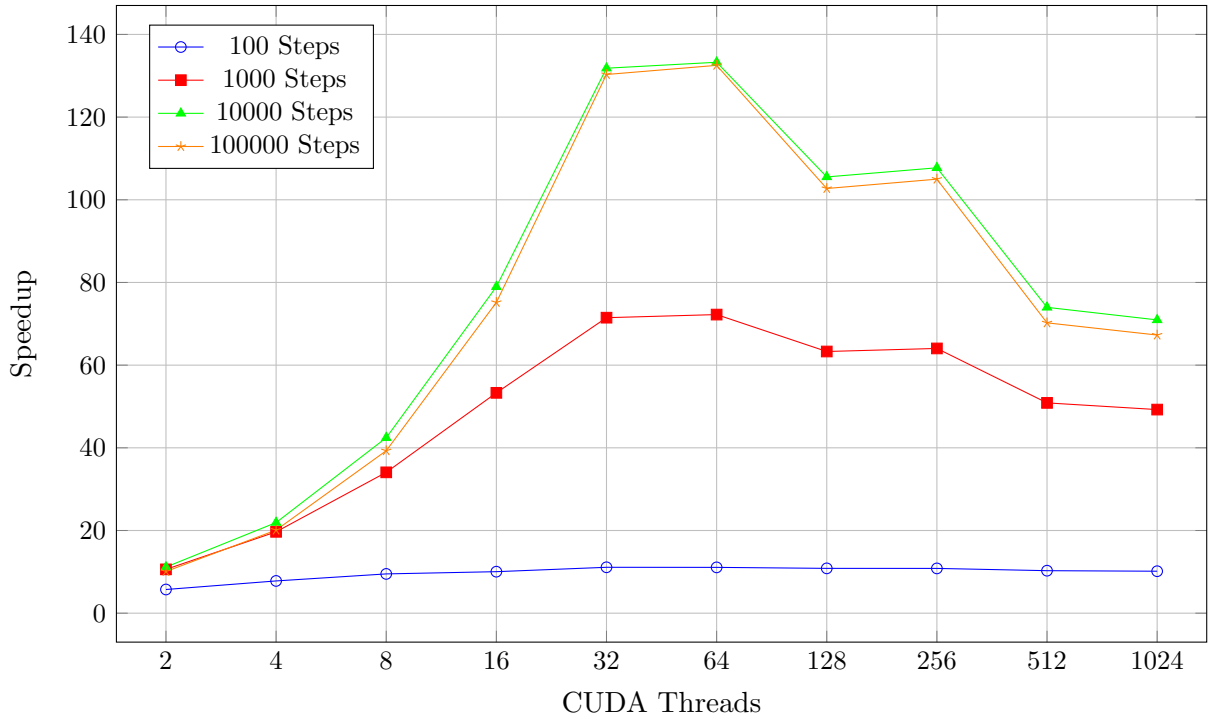

Figure 3: CUDA Speedup for 2000×2000 matrix

C.1.4 Speedup (OpenMP) in a 2000x2000 Matrix

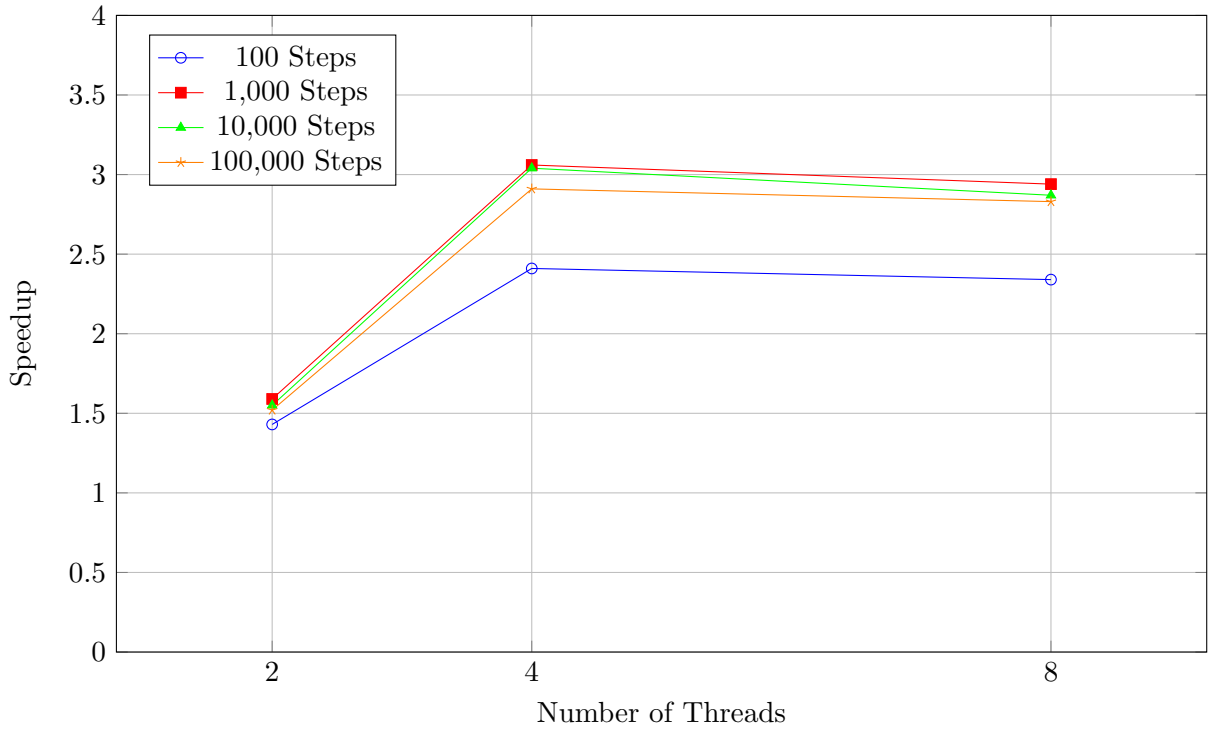

Figure 4: Speedup with OpenMP (2000x2000 matrix)

C.1.5 Speedup (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

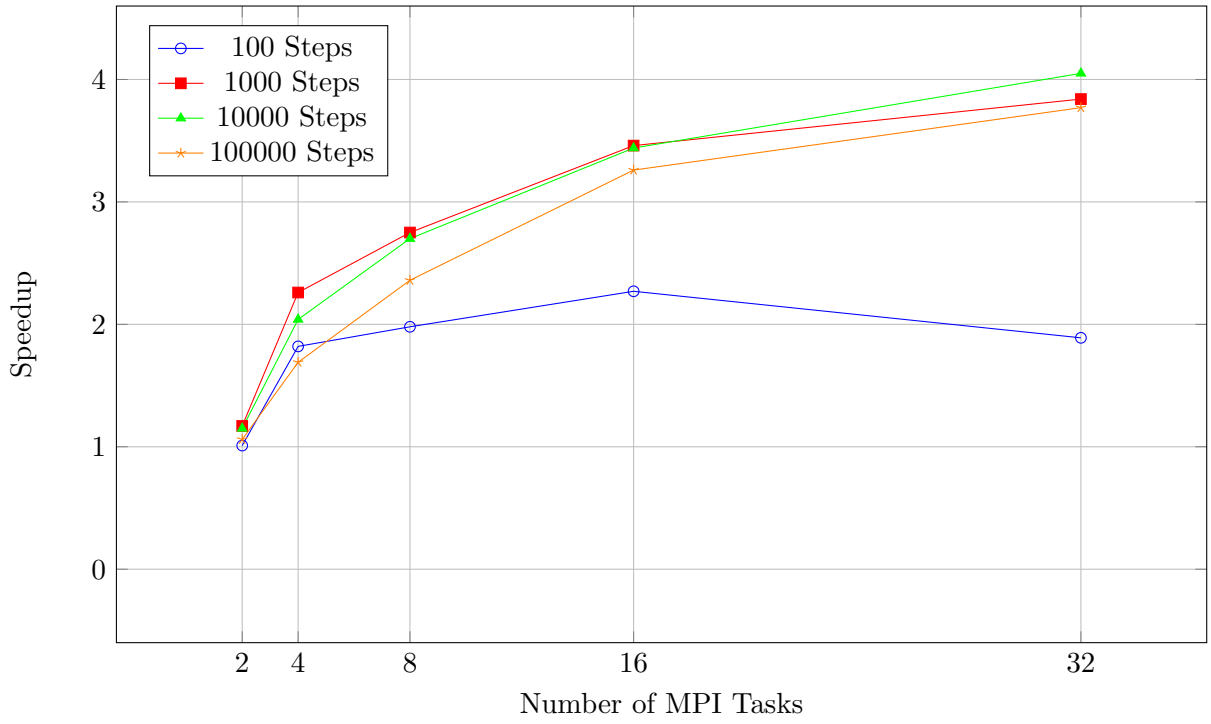

Figure 5: Speedup for OpenMP (2 threads) + MPI Tasks on 2000×2000 matrix

C.2 Efficiency

C.2.1 Efficiency (CUDA) in a 100x100 Matrix

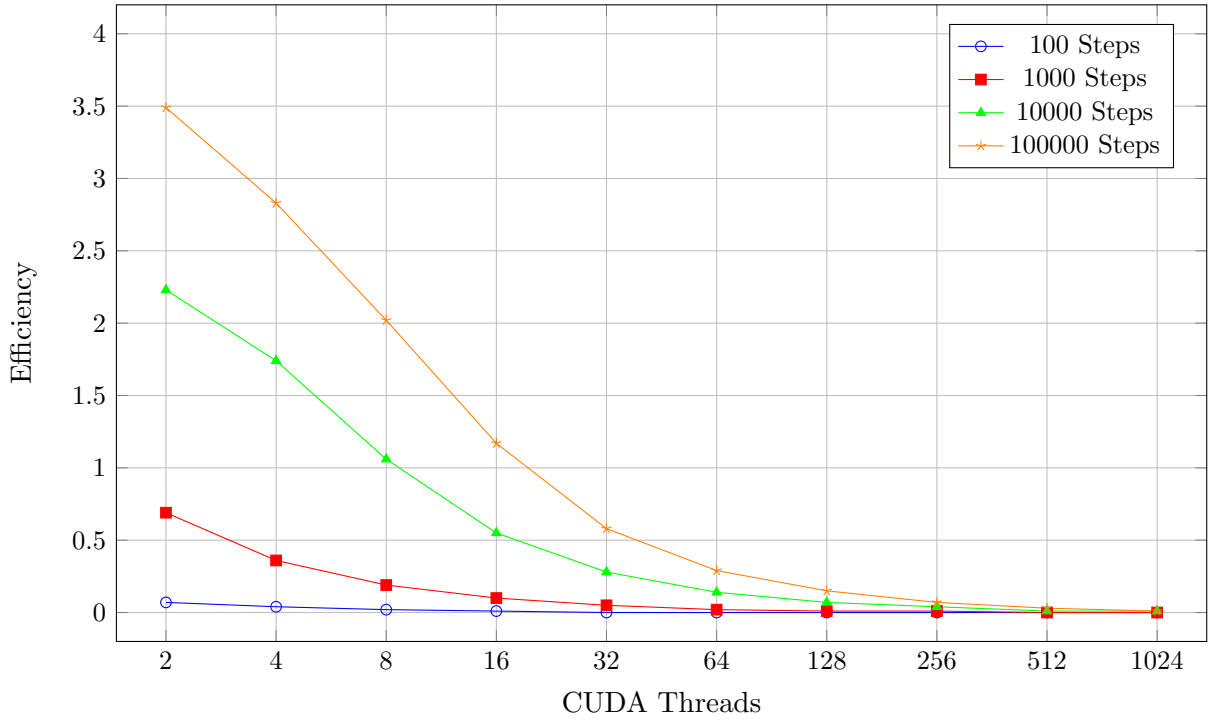

Figure 6: CUDA Efficiency for 100×100 matrix

C.2.2 Efficiency (CUDA) in a 1000x1000 Matrix

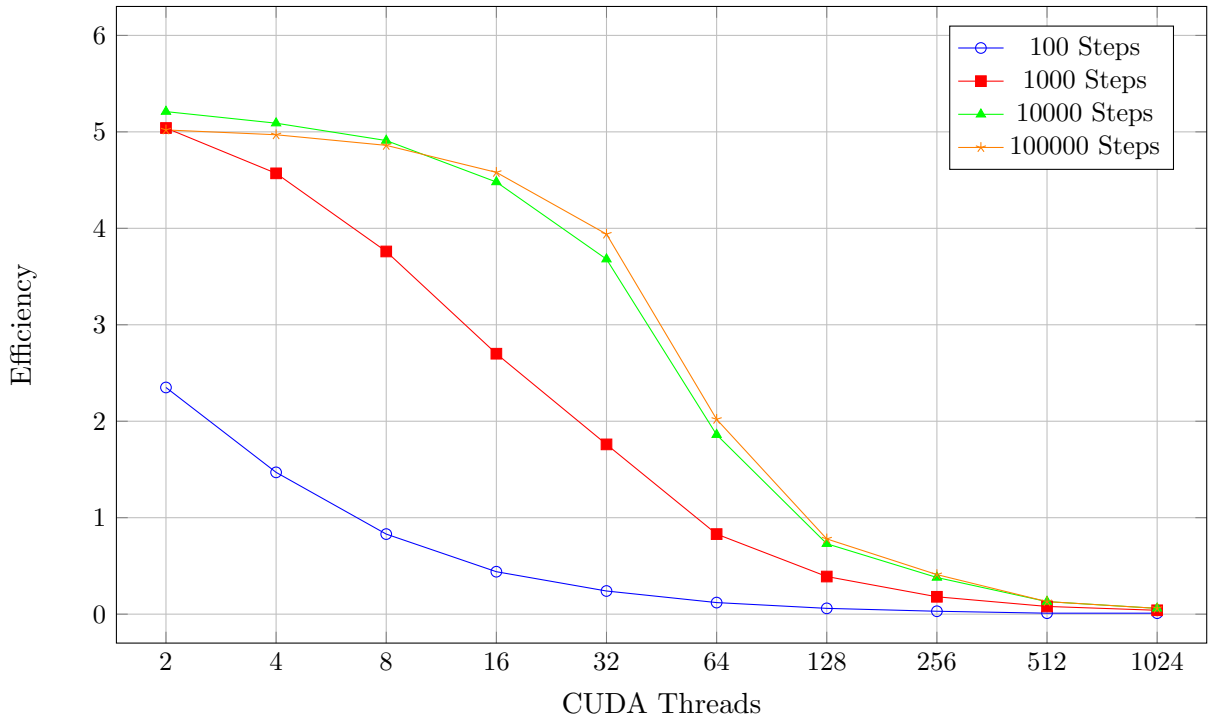

Figure 7: CUDA Efficiency for 1000×1000 matrix

C.2.3 Efficiency (CUDA) in a 2000x2000 Matrix

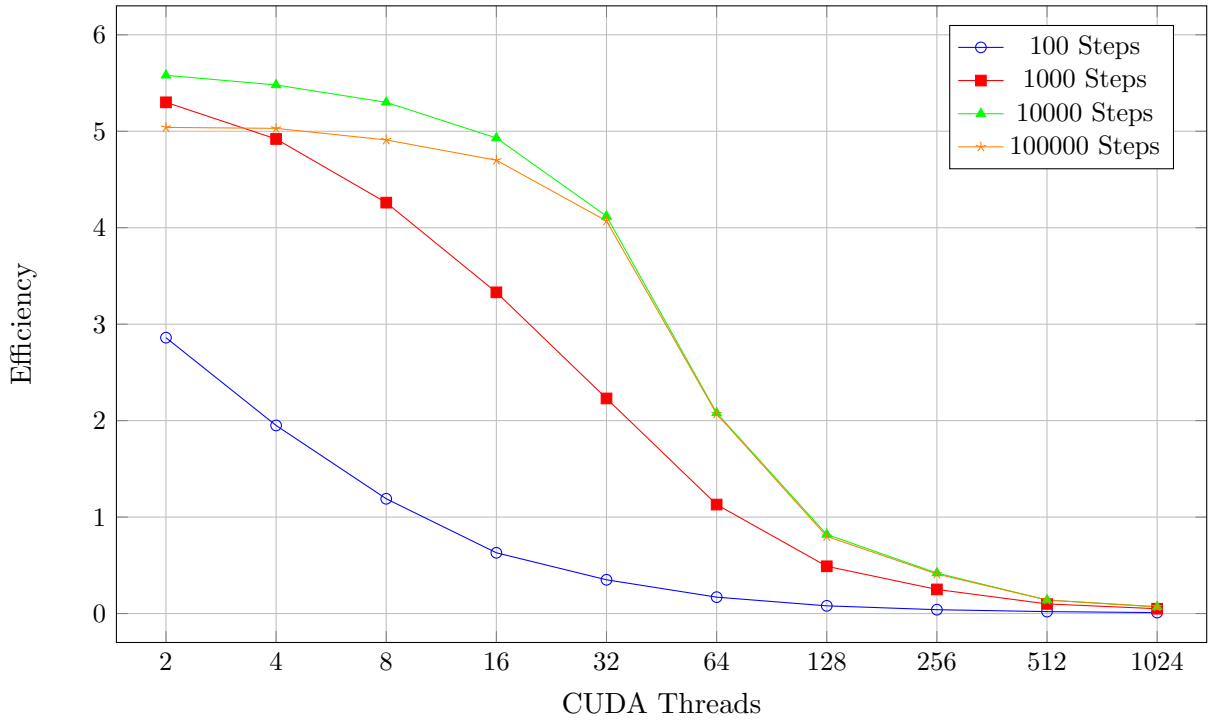

Figure 8: CUDA Efficiency for 2000×2000 matrix

C.2.4 Efficiency (OpenMP) in a 2000x2000 Matrix

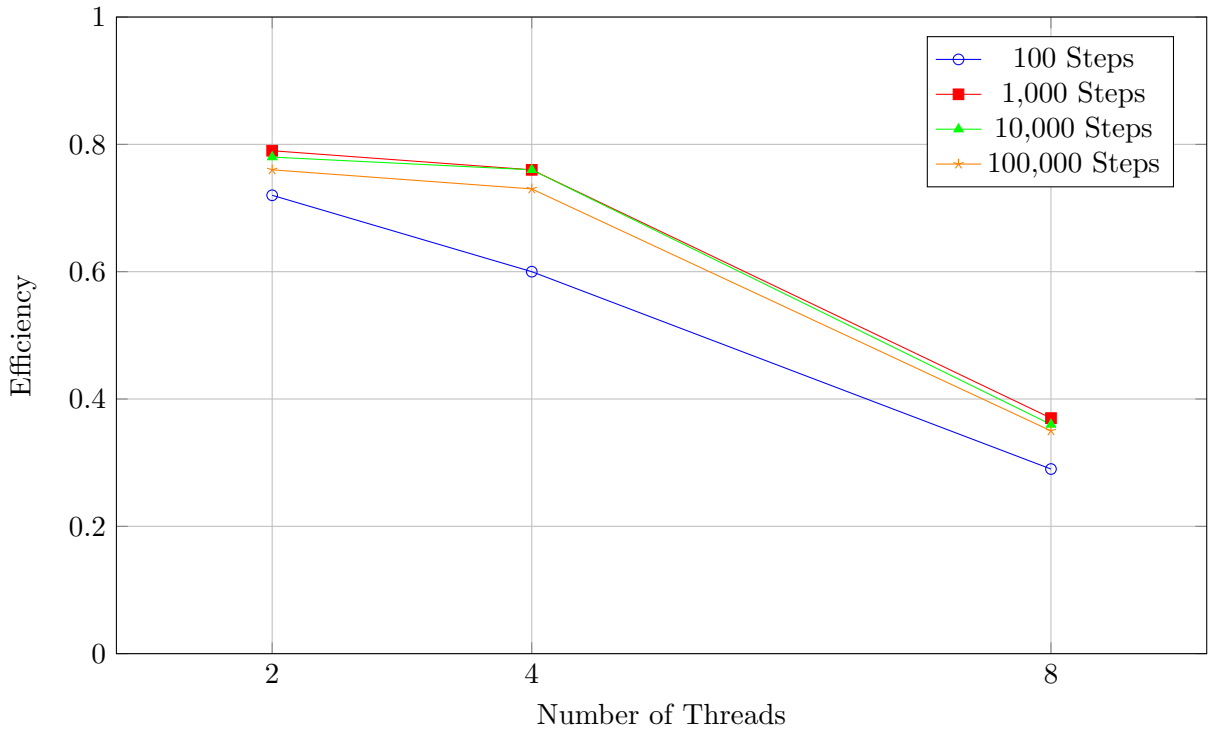

Figure 9: Efficiency with OpenMP (2000x2000 matrix)

C.2.5 Efficiency (2 OpenMP Threads + MPI Tasks) in a 2000x2000 Matrix

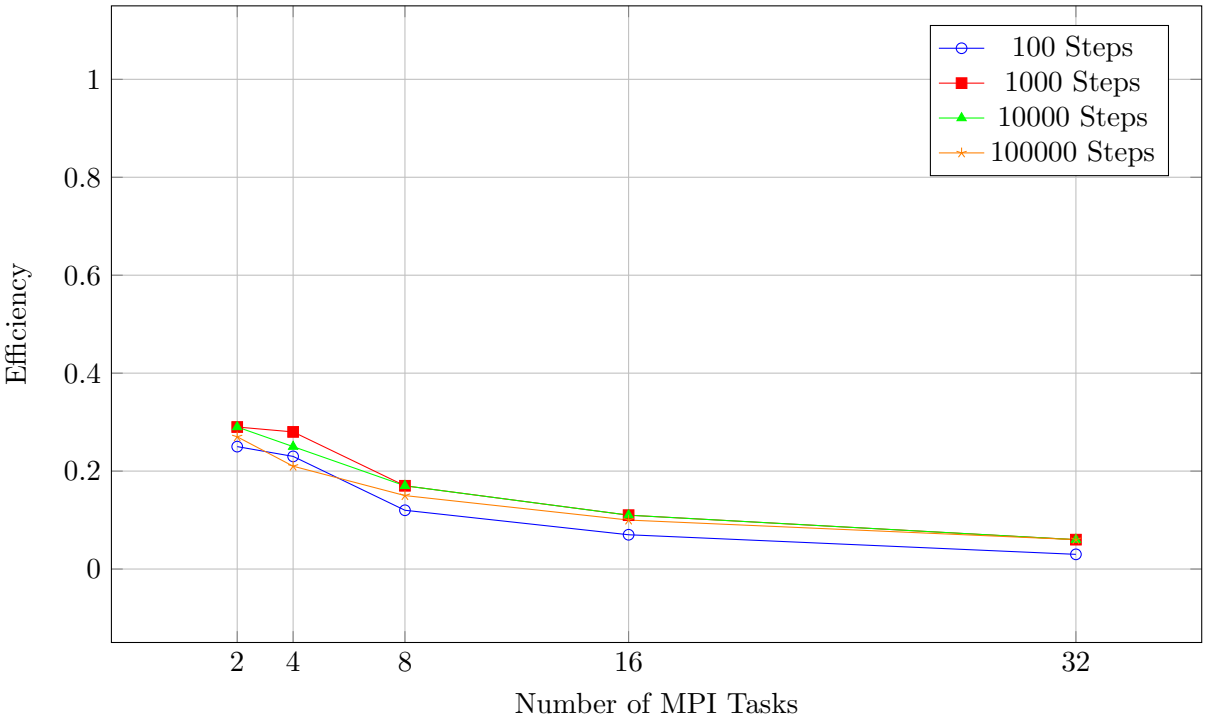

Figure 10: Efficiency for OpenMP (2 threads) + MPI Tasks on 2000×2000 matrix