Performance Evaluation of Parallel Matrix Exponentiation on CPUs and GPUs

A comprehensive study of threading strategies and their impact on computational efficiency using C and CUDA.

Andrew Struthers
May 14, 2024

CS 530: High Performance Computing
Dr. Szilard VAJDA

Department of Computer Science
Central Washington University
Ellensburg, WA, United States of America
Spring Quarter, 2024

Contents

Chapter 1

Parallel Matrix Multiplication Report

1.1 Problem Statement

The objective of this lab is to implement and evaluate the performance of calculating the Nth power of a square matrix of dimension M using advanced parallel programming techniques. Specifically, this lab will explore three distinct parallelization strategies in C: assigning one thread per matrix element, one thread per matrix row, and one thread per matrix column. Additionally, to provide a comprehensive comparison, the lab will also include an implementation of the matrix power operation using CUDA for GPU-based parallelization.

Matrix multiplication, though conceptually straightforward, becomes computationally intensive as the matrix size increases. The cost of this operation becomes apparent when a matrix is raised to a power, which involves multiple sequential matrix multiplications. Therefore, optimizing this operation through parallel computing is crucial for improving performance in high-performance computing (HPC) applications.

The primary objectives of this lab include the detailed implementation of matrix exponentiation using the aforementioned parallelization strategies in C, and assessing their computational complexity. By measuring and comparing the running times for different matrix sizes and powers, the lab aims to highlight the performance improvements achieved through parallelization. Furthermore, the lab will discuss the scalability and efficiency of each parallelization method, providing insights into their practical applications and limitations.

Parallel computing techniques offer significant performance enhancements by distributing the computational workload across multiple processing units. In this lab, the one-thread-per-element approach will maximize parallelism at the granularity of individual matrix elements, while the one-thread-per-row and one-thread-per-column strategies will exploit parallelism at coarser granularities. The CUDA implementation will leverage the massive parallelism capabilities of GPUs to further accelerate the computation.

1.2 Introduction

Matrix exponentiation is a fundamental operation in computational mathematics with applications across various fields, including physics, engineering, and computer science. It is often used in solving systems of linear equations, modeling dynamic systems, and performing transformations in graphics and data analysis. The direct computation of large matrix powers can be computationally expensive due to its polynomial complexity, making it a significant challenge in HPC.

1.2.1 Background and Motivation

Matrix multiplication, the core operation in matrix exponentiation, has a computational complexity of $O(M^3)$ for an $M \times M$ matrix when performed using the standard algorithm. As the size of the matrix and the exponent increase, the number of computations grows exponentially, leading to substantial execution times. This issue is particularly critical in applications requiring real-time processing or dealing with very large datasets.

Parallel computing offers a promising solution to mitigate the high computational cost associated with matrix exponentiation. By distributing the workload across multiple processors, parallel algorithms can significantly reduce execution times and improve efficiency. Modern computing architectures, such as multi-core CPUs and GPUs, provide the necessary hardware capabilities to implement these parallel algorithms effectively.

In this lab, students focus on parallelizing the matrix exponentiation process using different threading strategies in C and compare these with a GPU-based implementation using CUDA. The aim is to explore how parallelization can enhance performance and to identify the most efficient methods for large-scale matrix computations.

1.2.2 Objectives

The primary objectives of this lab are:

1. To implement matrix exponentiation using three parallelization strategies in C: one thread per matrix element, one thread per matrix row, and one thread per matrix column.
2. To develop a CUDA-based implementation of matrix exponentiation for comparison with the CPU-based methods. To evaluate the computational complexity of each parallelization strategy.
3. To measure and compare the performance improvements achieved through parallelization by analyzing running times for various matrix sizes and powers.
4. To discuss the scalability and efficiency of each parallelization method, providing insights into their practical applications and limitations.

By the end of this lab, students will have gained hands-on experience with parallel programming techniques and a deeper understanding of how to leverage modern computational resources for efficient matrix computations. This knowledge will be crucial for tackling more complex problems in future labs and real-world HPC applications.

1.3 Algorithm and Complexity Analysis

1.3.1 Matrix Multiplication Basics

Matrix multiplication is a key operation in matrix exponentiation. Given two matrices A and B of size $M \times M$, their product $C = A \times B$ is computed as:

$$C[i][j] = \sum_{k=0}^{M-1} A[i][k] \times B[k][j]$$

This operation has a time complexity of $O(M^3)$ when performed using the standard algorithm. The complexity arises from the triple nested loop required to compute each element of the resulting matrix. When raising a matrix A to the power N, the process involves performing this multiplication $N - 1$ times, leading to an overall time complexity of $O(N \times M^3)$.

1.3.2 Serial Algorithm Complexity

In a serial implementation, the matrix exponentiation operation involves iteratively multiplying the matrix by itself $N - 1$ times. The algorithm can be described as follows:

1. Initialize the result matrix R to the identity matrix of size $M \times M$.
2. For i from 1 to N:
   (a) Compute $R = R \times A$ using the standard matrix multiplication algorithm.

The serial algorithm's time complexity is dominated by the matrix multiplication step, resulting in an overall complexity of $O(N \times M^3)$. This makes the serial approach infeasible for large matrices or high powers due to the exponential growth in computation time.

1.3.3 Parallel Algorithm Design

To mitigate the high computational cost, we explore three parallelization strategies for matrix multiplication:

Thread per Element

In this approach, each element of the resulting matrix C is computed in a separate thread. The computation of each element $C[i][j]$ is independent, allowing for maximum parallelism. The algorithm can be described as:

1. For each element $C[i][j]$ in parallel:
   (a) Compute $C[i][j] = \sum_{k=0}^{M-1} A[i][k] \times B[k][j]$

The time complexity of this approach remains $O(N \times M^3)$, but the execution time is significantly reduced due to the concurrent computation of matrix elements.

Thread per Row

In this approach, each row of the resulting matrix C is computed in a separate thread. Each thread performs the matrix multiplication for one row, leveraging parallelism at the row level. The algorithm can be described as:

1. For each row i in parallel:
   (a) For each element j in the row:
      i. Compute $C[i][j] = \sum_{k=0}^{M-1} A[i][k] \times B[k][j]$

This method also maintains a time complexity of $O(N \times M^3)$ but benefits from parallel row processing, reducing execution time compared to the serial approach.

Thread per Column

In this approach, each column of the resulting matrix C is computed in a separate thread. Each thread performs the matrix multiplication for one column, exploiting parallelism at the column level. The algorithm can be described as:

1. For each column j in parallel:
   (a) For each element i in the column:
      i. Compute $C[i][j] = \sum_{k=0}^{M-1} A[i][k] \times B[k][j]$

Similar to the row-wise approach, the time complexity remains $O(N \times M^3)$, with the advantage of parallel column processing.

In all three parallelization strategies, the goal is to distribute the computational load across multiple threads, significantly reducing the overall execution time compared to the serial implementation. The actual performance gain depends on factors such as the number of available processing units and the efficiency of the threading model used.

1.4 Implementation Details

1.4.1 Serial Implementation in C

The serial implementation of matrix exponentiation involves iteratively multiplying the matrix by itself $N - 1$ times. The following steps were taken:

- **Data Structures:** Dynamic arrays were used to store the matrices. The matrices were represented as 2D arrays, allocated at runtime based on the dimension M.
- **Algorithm:** The matrix multiplication algorithm follows a straightforward triple nested loop structure to compute the product of two matrices.

Code Snippet:

```
void matrixMultiply(double **A, double **B, double **C, int M) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < M; j++) {
            C[i][j] = 0;
            for (int k = 0; k < M; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void matrixExponentiation(double **A, int N, int M) {
    double **result = allocateMatrix(M);
    initializeIdentityMatrix(result, M);

    for (int i = 1; i < N; i++) {
        double **temp = allocateMatrix(M);
        matrixMultiply(result, A, temp, M);
        copyMatrix(temp, result, M);
        freeMatrix(temp, M);
    }

    // Copy result to A or use result as needed
    freeMatrix(result, M);
}
```

1.4.2 Parallel Implementation in C

The parallel implementation explores three different strategies: one thread per element, one thread per row, and one thread per column. The pthreads library was used for managing threads. Below are the detailed implementations using a shared data structure.

Data Structure and Functions for Threading

Data Structure:

```
typedef struct {
    long double** base;
    long double** prev;
    long double** result;
    int dimension;
    int index;
} thread_data;
```

Thread Functions:

```
void* multiply_row(void* arg);
void* multiply_col(void* arg);
void* multiply_element(void* arg);
```

Thread per Element

In this approach, each thread computes a single element of the resulting matrix.

Algorithm 1: Thread per Element Multiplication

Data: base, prev, result matrices, dimension of matrices
Result: Updated result matrix
for $i \leftarrow 0$ ***to*** *dimension-1* **do**
  for $j \leftarrow 0$ ***to*** *dimension-1* **do**
    create_thread(*multiply_element*, {base, prev, result, dimension, i * dimension + j});
  end
end
join_all_threads();

Function:

```
void* multiply_element(void* arg) {
    thread_data* data = (thread_data*) arg;

    long double** base = data->base;
    long double** prev = data->prev;
    long double** result = data->result;

    int dimension = data->dimension;
    int row = data->index / dimension;
    int col = data->index % dimension;
    int k = 0;

    for(k = 0; k < dimension; k++) {
        result[row][col] += base[row][k] * prev[k][col];
    }

    pthread_exit(NULL);
}
```

Thread per Row

In this approach, each thread computes an entire row of the resulting matrix.

Algorithm 2: Thread per Row Multiplication

Data: base, prev, result matrices, dimension of matrices
Result: Updated result matrix
for $i \leftarrow 0$ ***to*** *dimension-1* **do**
  create_thread(*multiply_row*, {base, prev, result, dimension, i});
end
join_all_threads();

Function:

```
void* multiply_row(void* arg) {
    thread_data* data = (thread_data*) arg;

    long double** base = data->base;
    long double** prev = data->prev;
    long double** result = data->result;

    int dimension = data->dimension;
    int row = data->index;
    int col = 0, k = 0;

    for(col = 0; col < dimension; col++) {
        for(k = 0; k < dimension; k++) {
            result[row][col] += base[row][k] * prev[k][col];
        }
    }

    pthread_exit(NULL);
}
```

Thread per Column

In this approach, each thread computes an entire column of the resulting matrix.

Algorithm 3: Thread per Column Multiplication

Data: base, prev, result matrices, dimension of matrices
Result: Updated result matrix
for $j \leftarrow 0$ ***to*** *dimension-1* **do**
  create_thread(*multiply_col*, {base, prev, result, dimension, j});
end
join_all_threads();

Function:

```
void* multiply_col(void* arg) {
    thread_data* data = (thread_data*) arg;

    long double** base = data->base;
    long double** prev = data->prev;
    long double** result = data->result;
```

```
    int dimension = data->dimension;
    int col = data->index;
    int row = 0, k = 0;

    for(row = 0; row < dimension; row++) {
        for(k = 0; k < dimension; k++) {
            result[row][col] += base[row][k] * prev[k][col];
        }
    }

    pthread_exit(NULL);
}
```

1.4.3 CUDA Implementation

The CUDA implementation leverages the GPU's parallel processing capabilities to optimize matrix exponentiation. The approach involves three key steps: memory management, kernel execution, and synchronization.

Memory Management and Initialization

The implementation begins by allocating memory for the matrices and initializing them using a pseudo-random number generator (PRNG) provided by the cuRAND library. This setup is crucial for ensuring that each experiment starts with a consistent initial state.

Code Snippet:

```
void run_experiment(const int &dimension, const int &power) {
    for(const auto type : {ElementWise, RowWise, ColWise}) {
        for (int i = 0; i < 30; i++) {
            curandGenerator_t prng;
            build_curand_generator(prng, CURAND_RNG_PSEUDO_XORWOW, 1);

            allocate_memory(dimension);
            initialize_memory(dimension, prng);
            ...
            free_memory(dimension);
        }
    }
}
```

Matrix Multiplication Kernels

The core of the matrix exponentiation in CUDA is handled by three different kernels corresponding to the three parallelization strategies: element-wise, row-wise, and column-wise multiplication.

Kernel Functions:

- **Element-Wise Kernel:** This kernel is launched with a grid size equal to the dimension of the matrix and executes one thread per matrix element.
- **Row-Wise Kernel:** Launched with one block of threads, where each thread handles one row of the matrix multiplication.

- **Column-Wise Kernel:** Similar to the row-wise kernel but each thread handles one column of the matrix.

```
__host__ void raise_to_power(
    const long double *base,
    long double *previous,
    long double *current,
    const int& dimension,
    const int& power,
    const MatrixMultiplicationType& type)
{
    for(int i = 0; i < power; i++)
    {
        // Copy and clear memory
        for(int row = 0; row < dimension; row++)
        {
            cudaMemcpy(
                previous + row * dimension,
                current + row * dimension, sizeof(long double) * dimension,
                cudaMemcpyDeviceToDevice);
        }

        // Matrix multiplication
        switch(type)
        {
            case ElementWise:
                Multiply_ElementWise<<<dimension, dimension>>>
                (base, previous, current, dimension);
                break;
            case RowWise:
                Multiply_RowWise<<<1, dimension>>>
                (base, previous, current, dimension);
                break;
            case ColWise:
                Multiply_ColumnWise<<<1, dimension>>>
                (base, previous, current, dimension);
                break;
        }
        cudaDeviceSynchronize();
    }
}
```

Performance Measurement

To evaluate performance, each kernel execution is timed using high-resolution clocks before and after the kernel launch. The duration is recorded and averaged over 30 runs to ensure statistical significance.

Timing Code Snippet:

```
auto start = high_resolution_clock::now();

raise_to_power(dev_base, dev_previous, dev_current, dimension, power, type);
```

```
auto stop = high_resolution_clock::now();
auto duration = static_cast<double>(
    duration_cast<microseconds>(stop - start).count()) / 1e06;

printf("%d,%d,%s,%.6f\n",
    dimension,
    power,
    type == RowWise ? "Row wise" : type == ColWise ? "Column wise" : "Element wise",
    duration);
```

This section highlights the effectiveness of the CUDA implementation in optimizing the matrix exponentiation process through various parallelization strategies. It shows how different GPU parallelization techniques can be applied to enhance performance significantly compared to CPU-based implementations.

1.5 Experimental Setup

1.5.1 Hardware Configuration

The experiments were conducted on two different hardware setups:

- **CPU Configuration:** The serial and parallel implementations in C were tested on a system equipped with a 12th Gen Intel Core i9-12900 CPU clocked at 2.4GHz, with 64GB of RAM clocked at 4800 MHz.
- **GPU Configuration:** The CUDA implementation was tested on a system equipped with an NVIDIA RTX 3080 Ti Founders Edition.

1.5.2 Software Environment

The experiments utilized different software environments for C and CUDA implementations:

- **C Implementation:**
  - Operating System: Linux
  - Compiler: GCC
- **CUDA Implementation:**
  - Operating System: Windows
  - Compiler: NVCC (NVIDIA CUDA Compiler)
  - Toolkit: NVIDIA CUDA Toolkit
  - Build System: CMake

1.5.3 Testing Methodology

The performance of the matrix exponentiation algorithms was evaluated using various matrix dimensions and powers. The specific configurations tested are as follows:

- **Matrix Dimensions:** $\{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$
- **Matrix Powers:** $\{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000\}$

For each combination of matrix dimension and power, three different parallelization strategies were tested for the C implementation: element-wise, row-wise, and column-wise multiplication. Each test was conducted 30 times to ensure statistical significance, and the average execution time of these 30 runs was recorded. Experimentation consisted of varying the matrix dimension while keeping a fixed power number, as well as a different experiment consisting of having a fixed dimension while varying the matrix power.

Experimental Procedure

1. For each matrix dimension M and matrix power N:
   (a) Initialize the matrices and data structures.
   (b) Execute the serial matrix exponentiation algorithm and record the execution time.
   (c) Execute the parallel matrix exponentiation algorithms (element-wise, row-wise, and column-wise) and record the execution times.
   (d) Execute the CUDA matrix exponentiation algorithm and record the execution time.
   (e) Repeat each experiment 30 times.
   (f) Calculate the average execution time for each configuration.

The recorded execution times were then used to analyze and compare the performance of the different implementations and parallelization strategies.

1.5.4 Performance Metrics

The primary performance metric used in this study was the average execution time, measured in seconds. This metric was chosen to provide a clear indication of the computational efficiency of each algorithm under different configurations. Additionally, scalability and speedup factors were considered to evaluate the effectiveness of parallelization and the benefits of using GPU over CPU for matrix exponentiation.

1.6 Results and Discussion

1.6.1 Performance Metrics

The primary performance metric used in this study was the average execution time, measured in seconds, for the matrix exponentiation algorithms across various configurations of matrix dimensions and powers.

1.6.2 Serial vs Parallel Execution (C)

Element-wise Threading

The element-wise threading approach on the CPU showed significantly slower performance compared to both row-wise and column-wise threading. This was due to the overhead of creating and managing a large number of threads, each handling a single matrix element. The execution time increased rapidly with the matrix dimension, highlighting the inefficiency of this method for CPU parallelization.

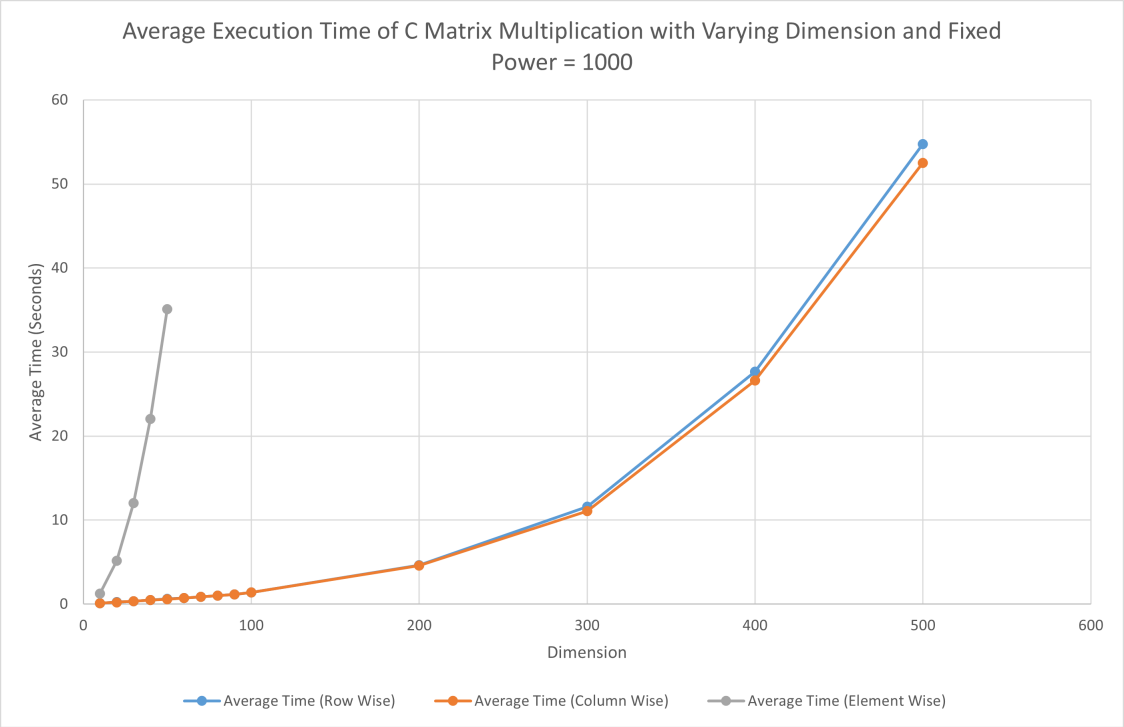

Figure 1.1: Graph of average execution time while varying dimension of computation in C.

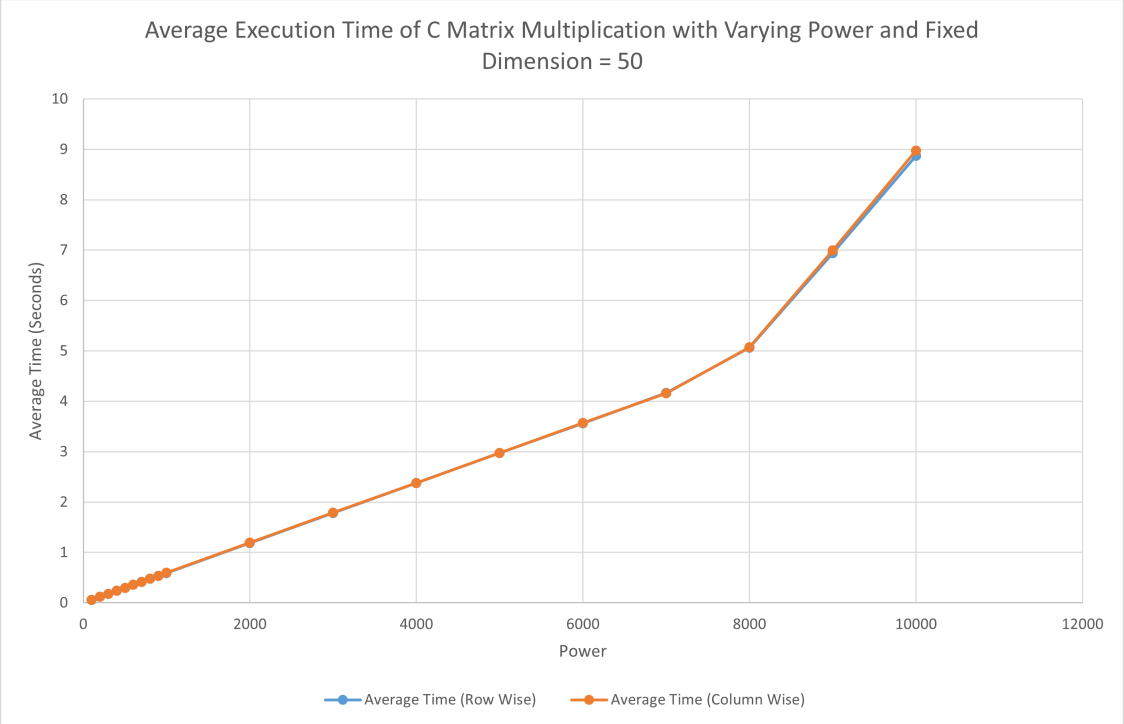

Figure 1.2: Graph of average execution time while varying power of computation in C.

Row-wise Threading

The row-wise threading approach demonstrated substantial performance improvements over the serial implementation. By assigning one thread per row, the computational load was effectively distributed, reducing the execution time. However, it was expected that row-wise threading would

perform better than column-wise due to the row-major order of C arrays, which should have provided better cache utilization.

Column-wise Threading

Contrary to expectations, the column-wise threading approach was slightly faster than row-wise threading on the CPU. This outcome suggests that factors other than memory layout, such as thread management overhead and specific implementation details, may have influenced the performance. Further investigation is needed to fully understand this behavior.

1.6.3 CUDA vs CPU Performance

The CUDA implementation was significantly faster than both the serial and parallel CPU implementations, primarily due to the massive parallelism offered by the GPU. The GPU's large number of cores allowed for efficient handling of element-wise operations, resulting in the fastest execution times even for large matrices.

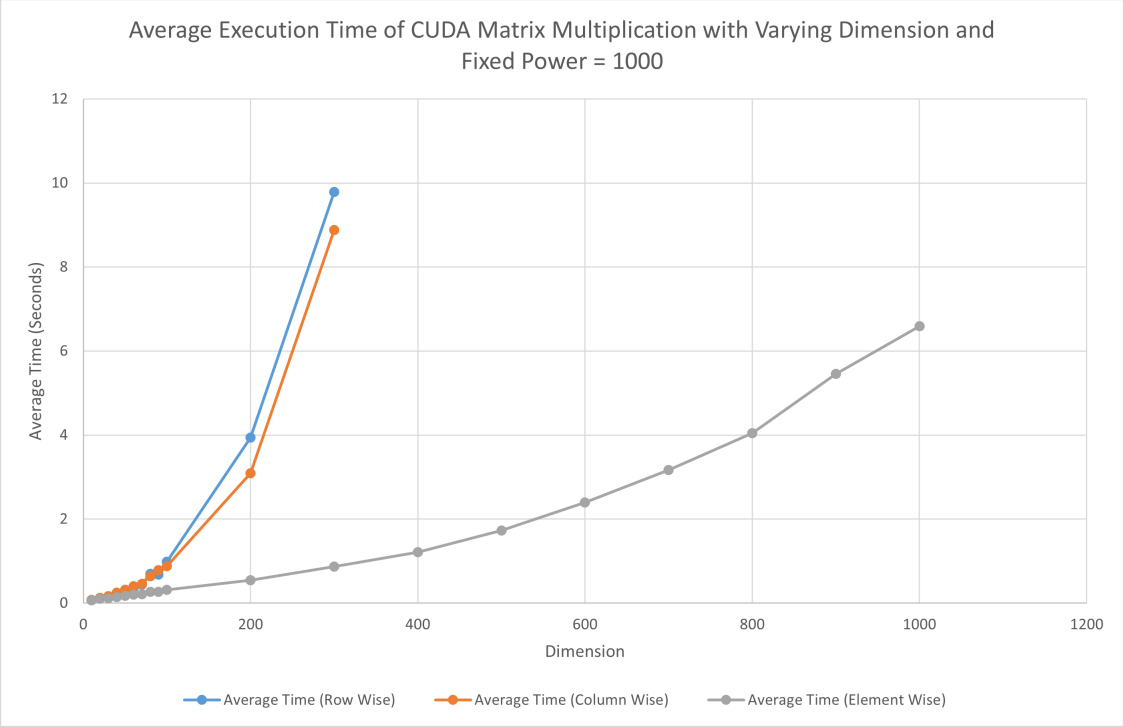

Figure 1.3: Graph of average execution time while varying dimension of computation in CUDA.

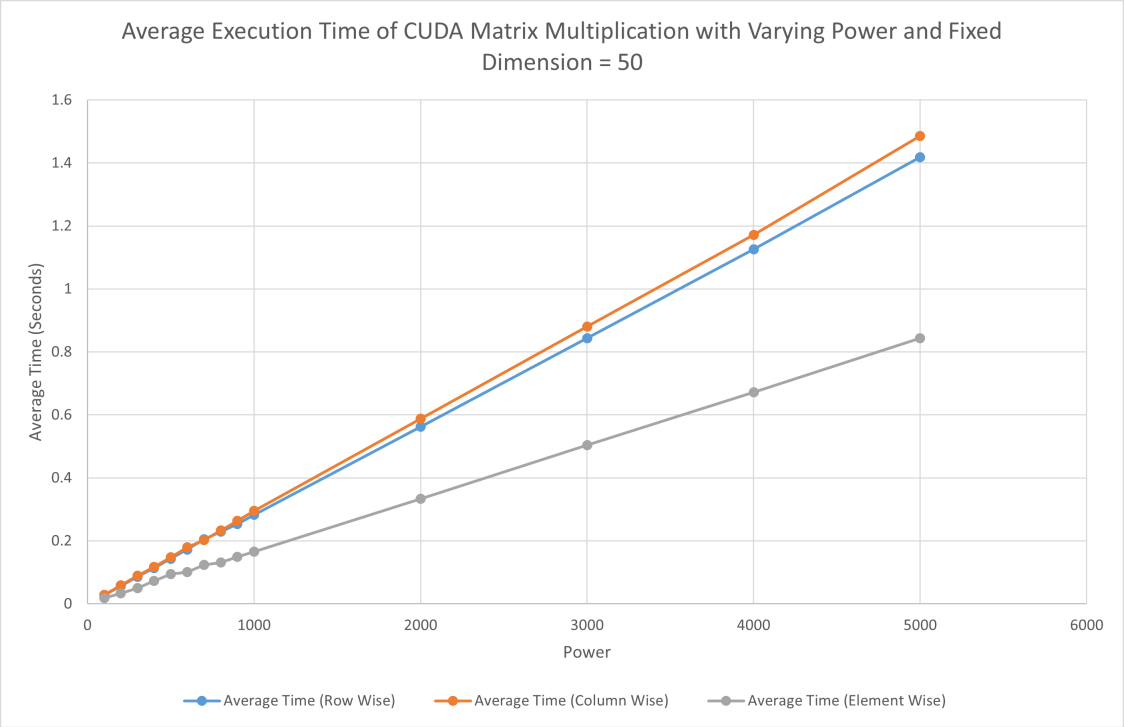

Figure 1.4: Graph of average execution time while varying power of computation in CUDA.

Element-wise Threading on GPU

Element-wise threading on the GPU outperformed all other methods, including row-wise and column-wise threading, regardless of matrix size. The GPU's architecture, with thousands of cores, is well-suited for fine-grained parallelism, making this approach highly efficient.

1.6.4 Scalability Analysis

Matrix Dimension Scaling

As the matrix dimension increased, the execution time scaled cubically ($O(M^3)$), consistent with the theoretical time complexity of matrix multiplication. This trend was observed across all implementations, both on the CPU and GPU.

Matrix Power Scaling

The execution time scaled linearly ($O(N)$) with the matrix power, which aligns with the expected complexity of matrix exponentiation. This linear relationship was evident in both the CPU and GPU implementations.

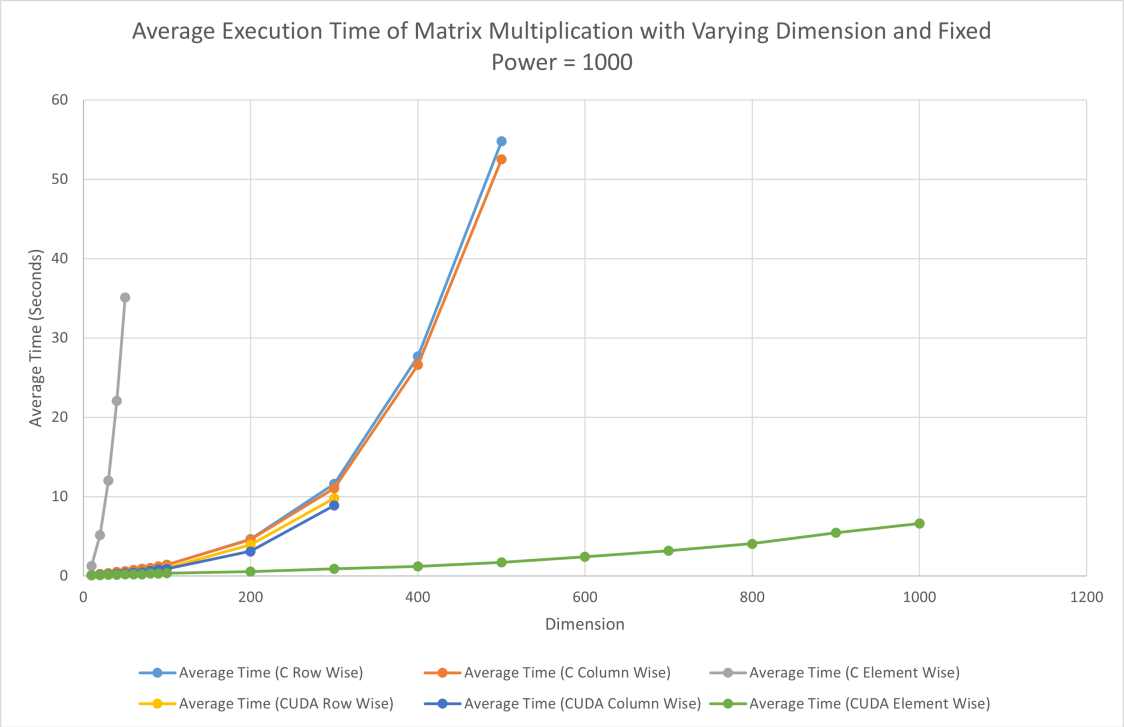

Figure 1.5: Graph of average execution time while varying dimension of computation in both C and CUDA.

Figure 1.6: Graph of average execution time while varying power of computation in both C and CUDA.

These figures clearly show the performance difference between CUDA and C in terms of row-wise, column-wise, and element-wise operations. It can be seen in Figure 1.5 that element-wise multiplication of a 1000×1000 matrix is faster than row- and column-wise multiplication of 300×300

matrices in C and CUDA. This pattern repeats in Figure 1.6, where it can be seen that CUDA element-wise operations are far faster than other operations of the same dimension and power. However, one interesting thing to note is that in Figure 1.5 we can see that C and CUDA both have very similar performance when it comes to row- and column-wise multiplication on matrices of the same dimension. Further experimentation could explore how this changes as the dimensions scale larger, as discussed below.

1.6.5 Discussion of Results

The results highlight the advantages of parallel computing for matrix exponentiation, with significant speedups observed in both CPU and GPU implementations compared to the serial approach. The GPU's performance was particularly impressive, demonstrating the power of modern GPUs for computationally intensive tasks.

The unexpected performance of column-wise threading on the CPU suggests that memory layout is not the sole determinant of performance. Other factors, such as thread scheduling and synchronization overhead, likely played a role. This finding underscores the complexity of optimizing parallel algorithms and the need for careful consideration of all aspects of the implementation.

Overall, the CUDA implementation's superior performance reaffirms the value of GPU computing for high-performance tasks. The results also emphasize the importance of choosing the right parallelization strategy, as the effectiveness of different approaches can vary significantly based on the underlying hardware and specific problem characteristics.

1.7 Conclusion

In this lab, we explored various parallelization strategies for matrix exponentiation using C and CUDA. The experiments demonstrated significant performance gains achievable through parallel computing, with CUDA showing the most impressive improvements. The detailed findings are summarized as follows:

1.7.1 Summary of Findings

- **Serial vs Parallel Execution (C):** The parallel implementations in C were significantly faster than the serial approach. Both row-wise and column-wise threading provided substantial speedups, with column-wise threading slightly outperforming row-wise threading. Element-wise threading on the CPU, however, was inefficient due to the high overhead of managing a large number of threads.
- **CUDA vs CPU Performance:** The CUDA implementation vastly outperformed all CPU implementations, demonstrating the power of GPU parallelism. The element-wise threading strategy on the GPU was the most efficient, leveraging the massive number of GPU cores to handle fine-grained parallel tasks effectively.
- **Scalability:** The execution time for matrix exponentiation scaled cubically with matrix dimension ($O(M^3)$) and linearly with matrix power ($O(N)$), consistent with theoretical expectations. This scalability was observed across both CPU and GPU implementations, confirming the correctness of the algorithms and their implementation.

1.7.2 Implications for High-Performance Computing

The results of this lab highlight the critical role of parallel computing in optimizing computationally intensive tasks like matrix exponentiation. The significant speedups achieved with parallel

implementations underscore the importance of leveraging multi-core CPUs and GPUs for high-performance applications. The findings also demonstrate the need to carefully select and optimize parallelization strategies based on the specific hardware architecture and problem characteristics.

1.7.3 Discussion of Unexpected Results

One of the surprising outcomes of this lab was the superior performance of column-wise threading over row-wise threading on the CPU. Given that C arrays are stored in row-major order, it was anticipated that row-wise threading would be more efficient due to better cache utilization. This discrepancy suggests that other factors, such as thread management overhead and specific implementation details, can significantly impact performance. Further investigation is needed to fully understand this behavior, which could involve profiling the code to identify bottlenecks and inefficiencies.

1.7.4 Consideration of Thread Pools in C

Thread pools were implemented in the C parallelization code and were shown to provide faster timings than the non-thread-pool implementation; however, the results depicted in this report did not include the thread-pool version of the C code. This was done to maintain as fair of a comparison between the C and CUDA implementations as possible. While thread pools can significantly reduce the overhead associated with thread creation and management by reusing a fixed set of threads, CUDA does not inherently support thread pools due to its different architecture and execution model. By avoiding the use of thread pools in the C implementation reported, we aimed to provide a clearer comparison of the raw parallel processing capabilities of the CPU and GPU.

The decision to exclude thread pools was made to focus on the core parallelization techniques and their direct impact on performance. This approach ensures that the observed performance differences between the C and CUDA implementations are primarily attributable to the inherent architectural advantages of the GPU, rather than optimizations specific to the CPU environment. Future work could include exploring the impact of thread pools on the performance of C implementations, providing additional insights into the optimization of parallel algorithms for multi-core CPUs.

1.7.5 Future Work

Building on the findings of this lab, future work can focus on several areas to further enhance the understanding and performance of matrix exponentiation algorithms:

- **Optimization of Thread Management:** Investigate and optimize the overhead associated with thread creation and management, particularly for element-wise threading on the CPU.
- **Advanced Parallelization Techniques:** Explore more sophisticated parallelization techniques, such as hybrid models combining row-wise and column-wise threading, or utilizing advanced libraries like OpenMP and MPI for better performance scaling.
- **Profiling and Bottleneck Analysis:** Conduct detailed profiling of the parallel implementations to identify and address performance bottlenecks, ensuring that memory access patterns and cache utilization are optimized.
- **Application to Larger Datasets:** Extend the experiments to even larger matrices and higher powers to evaluate the scalability and efficiency of the algorithms in more demanding scenarios.
- **Cross-Platform Comparison:** Compare the performance of the implementations across different platforms and hardware configurations to gain broader insights into the portability

and efficiency of the parallelization strategies.

1.7.6 Conclusion

In conclusion, this lab provided valuable insights into the benefits and trade-offs of different parallelization strategies for matrix exponentiation. The significant performance gains achieved with CUDA and the nuanced performance characteristics of CPU parallelization underscore the importance of optimizing algorithms for specific hardware architectures. The decision not to use thread pools in the C implementation ensured a fair comparison with CUDA, highlighting the inherent advantages of GPU computing. The results and observations from this lab lay a strong foundation for future exploration and optimization of high-performance computing techniques.