Optimization of Single Threaded Code

A study of High Performance Computing techniques to optimize and improve on runtimes of single-threaded code without affecting algorithmic complexity.

Andrew Struthers

May 31, 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

Optimized Matrix Multiplication Report

1.1 Introduction

1.1.1 Problem Statement

The goal of this lab is to implement and assess the performance of calculating the Nth power of a square matrix of dimension M using different high-performance programming optimization techniques. The primary objectives include timing algorithms for matrix exponentiation, evaluating their computational complexity, and visualizing the running times for varying matrix sizes and powers. We are to compare the execution time, considering the same matrix size and power as previously used, when optimizing the code using automatic optimization with compiler flags, and manually optimized code using techniques learned in class.

1.1.2 Background and Motivation

Matrix exponentiation is a fundamental operation in various scientific and engineering applications, including computer graphics, numerical simulations, and machine learning. The computational complexity of matrix multiplication, which underpins matrix exponentiation, is $O(M^3)$, making it a time-consuming task for large matrices. This complexity poses a significant challenge, especially in high-performance computing (HPC) environments where efficiency and speed are paramount.

Optimization techniques in programming can significantly reduce the execution time of computationally intensive tasks. Compiler optimization flags, efficient memory access patterns, and the use of register variables are some of the techniques that can enhance performance. Understanding and applying these techniques are crucial for anyone working in fields that require intensive computation.

This lab aims to explore these optimization techniques in the context of matrix exponentiation. By comparing different implementations and optimization strategies, we can identify the most effective methods for improving performance.

1.1.3 Objectives

The objectives of this lab are as follows:

- Implement a matrix exponentiation algorithm using 2D dynamic arrays in C.
- Evaluate the performance of the algorithm using different compiler optimization flags (O1, O2, O3, Ofast, Os).
- Optimize the algorithm by flattening the 2D dynamic arrays into 1D arrays to improve memory access patterns.

- Further optimize the algorithm by using register variables to reduce memory traffic and leverage faster access speeds.
- Compare the execution times of the original and optimized implementations for varying matrix sizes and powers.
- Analyze the performance improvements achieved through different optimization techniques.
- Provide a comprehensive analysis and discussion of the results, highlighting the most effective optimization strategies.

By the end of this lab, students will have gained hands-on experience with optimization techniques for high-performance computing 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.

1.2 Algorithm and Complexity Analysis

1.2.1 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 $O(N \times M^3)$.

1.2.2 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.

1.2.3 Optimization Techniques

To improve the performance of matrix exponentiation, several optimization techniques can be applied:

- **Compiler Optimization Flags:** Modern compilers offer various optimization flags that can automatically improve the performance of code. Common flags include O1, O2, O3, Ofast, and Os, each providing different levels of optimization.

- **Flattened Arrays:** Converting 2D dynamic arrays into 1D arrays can enhance memory access patterns by ensuring contiguous memory allocation, thereby reducing cache misses and improving performance.
- **Register Variables:** Using register variables for frequently accessed data can significantly reduce memory access times, as registers are much faster than RAM.
- **Loop Unrolling:** Manually unrolling loops can decrease the overhead of loop control and increase the instruction-level parallelism, leading to better performance.

These optimization techniques are crucial for high-performance computing, where maximizing computational efficiency is essential. In this lab, we will explore the impact of these techniques on the performance of matrix exponentiation.

1.3 Implementation Details

1.3.1 Standard Implementation in C

The initial implementation of the matrix exponentiation algorithm uses 2D dynamic arrays in C. The key functions include initializing the matrix, raising it to a power, and performing matrix multiplication. Below are the code snippets for the main components:

```c
int main(int argc, char** argv)
{
    int num_dimensions = 14;
    int num_powers = 19;
    int dimensions[14] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200,
       300, 400, 500};
    int powers[19] = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000,
       2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};

    for(int i = 0; i < num_dimensions; i++)
    {
        run_experiment(dimensions[i], 1000);
    }

    for(int i = 0; i < num_powers; i++)
    {
        run_experiment(50, powers[i]);
    }

    return 0;
}
```

Listing 1.1: Main Function for Serial Implementation

```c
void run_experiment(int matrix_dimension, int power)
{
    for(int i = 0; i < 30; i++)
    {
        allocate_memory(matrix_dimension);

        srand(time(NULL));
        initialize_matrix(Random, matrix_dimension);

        struct timeval timeofday_start, timeofday_end;
        double timeofday_elapsed;

        gettimeofday(&timeofday_start, NULL);

        raise_to_power(matrix_dimension, power, 0);

        gettimeofday(&timeofday_end, NULL);

        timeofday_elapsed = (timeofday_end.tv_sec -timeofday_start.tv_sec)
             + (timeofday_end.tv_usec - timeofday_start.tv_usec) /
            1000000.0;

        printf("%d,%d,%.6f,Standard\n", matrix_dimension, power,
            timeofday_elapsed);

        free_memory(matrix_dimension);
    }
}
```

Listing 1.2: Run Experiment Function

```c
int multiply_matrix(int dimension)
{
    int all_nan = 1;
    for(int i = 0; i < dimension; i++)
    {
        for(int j = 0; j < dimension; j++)
        {
            for(int k = 0; k < dimension; k++)
            {
                current[i][j] += previous[i][k] * matrix[k][j];
            }
            if(isnormal(current[i][j]))
            {
                all_nan = 0;
            }
        }
    }
    if(all_nan)
    {
        printf("All entries in matrix are NaN-like\n");
        return 1;
    }
    return 0;
}
```

Listing 1.3: Matrix Multiplication Function

1.3.2 Optimized Implementation in C

The optimized implementation involves flattening the 2D dynamic arrays into 1D arrays and using register variables. Below are the code snippets for the main components of the optimized implementation:

```c
int main(int argc, char** argv)
{
    int num_dimensions = 14;
    int num_powers = 19;
    int dimensions[14] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200,
        300, 400, 500};
    int powers[19] = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000,
       2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};

    for(int i = 0; i < num_dimensions; i++)
    {
        run_experiment(dimensions[i], 1000);
    }

    for(int i = 0; i < num_powers; i++)
    {
        run_experiment(50, powers[i]);
    }

    return 0;
}
```

Listing 1.4: Main Function for Flattened Implementation

```c
void run_experiment(int matrix_dimension, int power)
{
    for(int i = 0; i < 30; i++)
    {
        allocate_memory(matrix_dimension);

        srand(time(NULL));
        initialize_matrix(Random, matrix_dimension);

        struct timeval timeofday_start, timeofday_end;
        double timeofday_elapsed;

        gettimeofday(&timeofday_start, NULL);

        raise_to_power(matrix_dimension, power, 0);

        gettimeofday(&timeofday_end, NULL);

        timeofday_elapsed = (timeofday_end.tv_sec -timeofday_start.tv_sec)
             + (timeofday_end.tv_usec - timeofday_start.tv_usec) /
            1000000.0;

        printf("%d,%d,%.6f,Flattened\n", matrix_dimension, power,
            timeofday_elapsed);

        free_memory(matrix_dimension);
    }
}
```

Listing 1.5: Run Experiment Function with Flattened Arrays

```c
int multiply_matrix(int dimension)
{
    int all_nan = 1;
    for(int i = 0; i < dimension; i++)
    {
        for(int j = 0; j < dimension; j++)
        {
            int matrix_index = i * dimension + j;
            for(int k = 0; k < dimension; k++)
            {
                current[matrix_index] += previous[i * dimension + k] *
                   matrix[k * dimension + j];
            }
            if(isnormal(current[matrix_index]))
            {
                all_nan = 0;
            }
        }
    }
    if(all_nan)
    {
        printf("All entries in matrix are NaN-like\n");
        return 1;
    }
    return 0;
}
```

Listing 1.6: Matrix Multiplication Function with Flattened Arrays

```c
int multiply_matrix(register int dimension)
{
    register int all_nan = 1;
    for(register int i = 0; i < dimension; i++)
    {
        for(register int j = 0; j < dimension; j++)
        {
            register int matrix_index = i * dimension + j;
            register MATRIX_TYPE temp = 0.0;
            register MATRIX_TYPE left = 0.0;
            register MATRIX_TYPE right = 0.0;

            for(register int k = 0; k < dimension; k++)
            {
                left = previous[i * dimension + k];
                right = matrix[k * dimension + j];
                temp += left * right;
            }
            current[matrix_index] = temp;
            if(isnormal(current[matrix_index]))
            {
                all_nan = 0;
            }
        }
    }
    if(all_nan)
    {
        printf("All entries in matrix are NaN-like\n");
        return 1;
    }
    return 0;
}
```

Listing 1.7: Matrix Multiplication Function with Register Variables

1.4 Experimental Setup

1.4.1 Hardware Configuration

The experiments were conducted on a system equipped with the following hardware:

- **CPU:** 12th Gen Intel Core i9-12900 CPU @ 2.4GHz
- **RAM:** 64GB @ 4800 MHz

1.4.2 Software Environment

The software environment used for the experiments included:

- **Operating System:** Linux
- **Compiler:** GCC (GNU Compiler Collection)

1.4.3 Testing Methodology

Experimental Procedure

The testing methodology involves running a series of experiments to measure the execution time of matrix exponentiation for different matrix sizes and powers, using both the original and optimized implementations. The procedure included the following steps:

1. Implement the matrix exponentiation algorithm using 2D dynamic arrays in C.
2. Compile the code with different compiler optimization flags: O1, O2, O3, Ofast, and Os.
3. Run the experiments for each matrix dimension and power combination.
4. Record the execution time for each run.
5. Optimize the implementation by flattening the 2D arrays into 1D arrays.
6. Further optimize the code by using register variables.
7. Repeat the experiments with the optimized code, both with and without the most effective compiler optimization flag (O3).
8. Compare the execution times of the original and optimized implementations.

1.4.4 Performance Metrics

The primary performance metric used in this study was the execution time, measured in seconds. The experiments were conducted for the following matrix dimensions and powers:

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

The experiments were repeated 30 times for each combination of matrix dimension and power to ensure the reliability of the results. The average execution time was calculated and used for analysis. The experiments aim to evaluate:

- **Effectiveness of Compiler Optimization Flags:** The impact of different compiler optimization levels (O1, O2, O3, Ofast, Os) on the execution time.
- **Impact of Flattened Arrays:** The performance improvement achieved by converting 2D dynamic arrays to 1D arrays.

- **Impact of Register Variables:** The performance improvement achieved by using register variables for frequently accessed data.
- **Scalability:** The scalability of the implementations with increasing matrix dimensions and powers.

1.5 Results and Discussion

1.5.1 Performance Metrics

The primary performance metric used in this study was the execution time, measured in seconds. The experiments were conducted for varying matrix dimensions and powers, and the results were recorded for different optimization techniques.

1.5.2 Compiler Optimization Comparison

The first set of experiments compared the execution times of the original matrix exponentiation implementation using various compiler optimization flags: O1, O2, O3, Ofast, and Os.

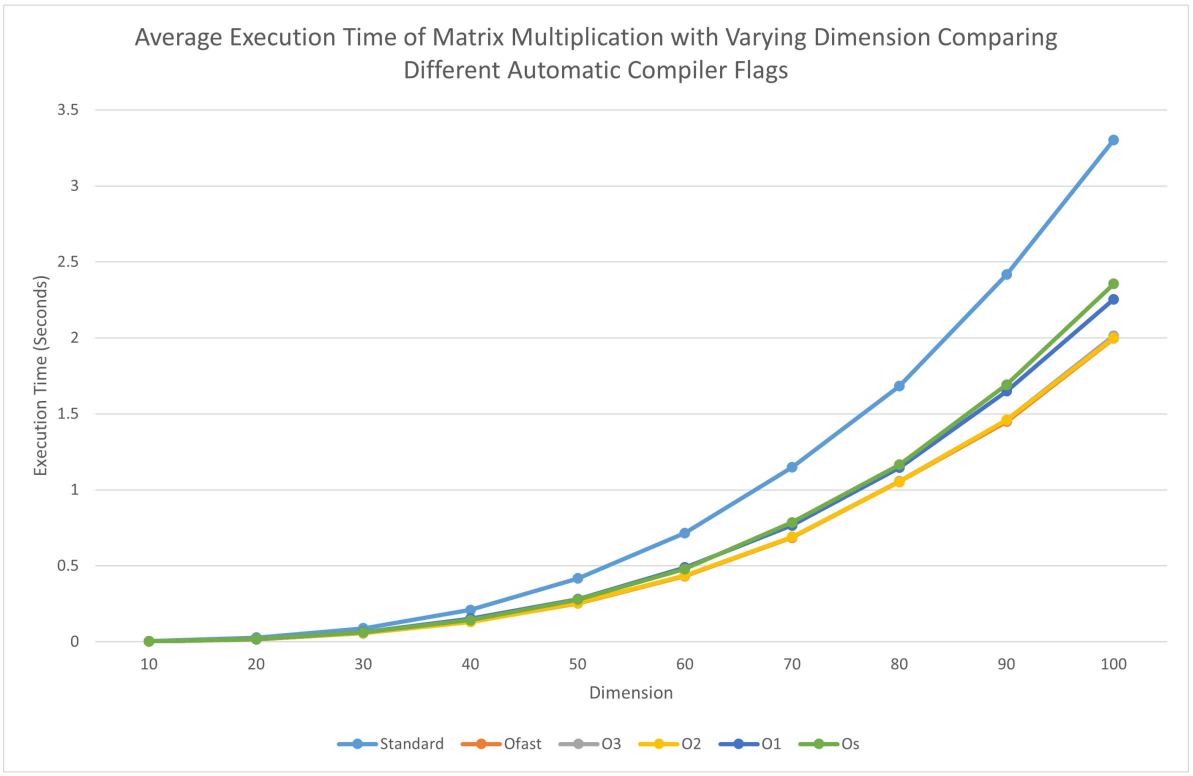

Figure 1.1: Execution times for varying matrix dimensions using different compiler optimization flags.

Here, we can see that the O2 and O3 speeds are almost exactly the same. This trend continues for the following figures as well. A more detailed explanation can be found below.

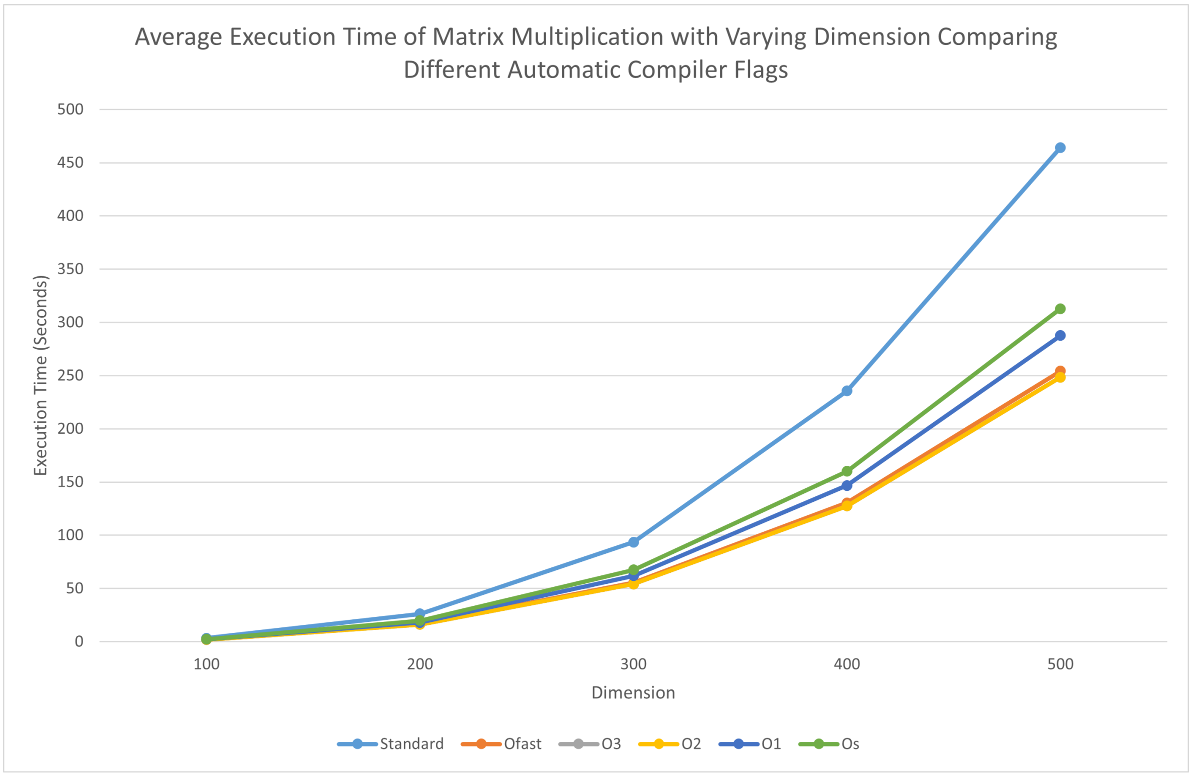

Figure 1.2: Execution times for varying matrix dimensions using different compiler optimization flags.

Once again, O2 and O3 are performing incredibly similarly, but both of them are faster than every other large dimension scaling problem. However, if we look at just the differences between the O2 speeds and the O3 speeds, we can see that O3 is actually producing faster times for high dimension operations:

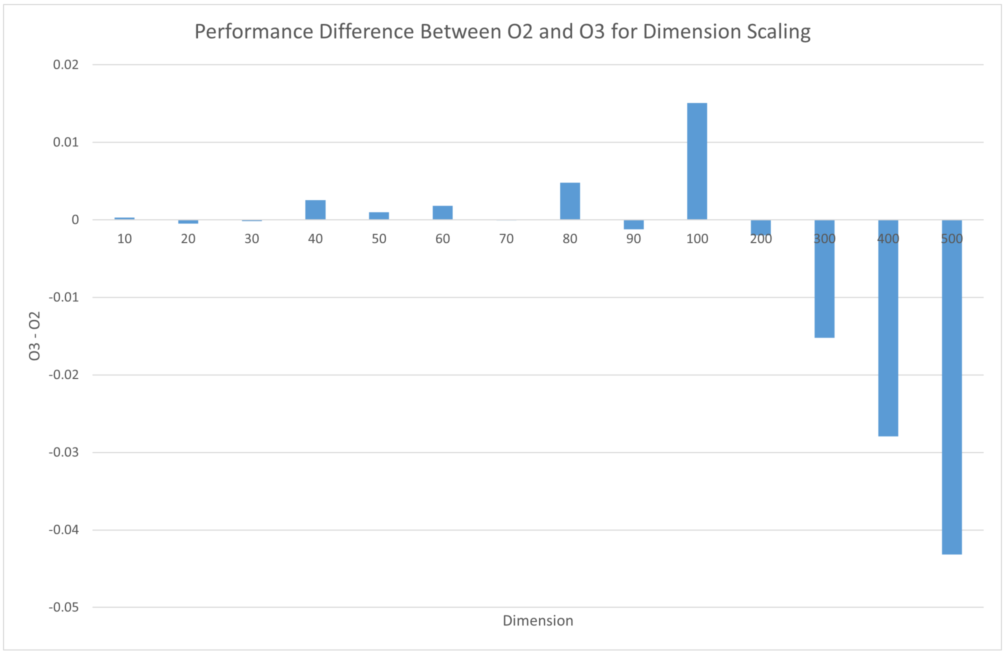

Figure 1.3: Difference in execution times between O2 and O3 optimization. Positive values mean O2 is faster, negative values mean O3 is faster.

We can see that O3 optimization is marginally faster than O2 for the large dimensions. For this reason, it was decided that the best optimization flag for this project would be the O3 flag, even though Ofast and O2 are, on average, faster than O3. This result is discussed below, because it is not what was expected. Due to the demanding nature of very large dimension matrices, however, O3 was considered to be the best.

The trend described above can be seen once again when comparing the power scaling with different optimization flags. Notice that in large powers, the time levels off. This is due to matrices, despite being initialized with values between $[-1.0, 1.0]$, would be filled with $-\inf, +\inf, -NaN$, or $+NaN$. The matrix multiplication code had an early return condition if this was the case. As shown in the figures, the matrix multiplication time plateaued at A^{8000}, which is when the $NaN-$like early stopping conditions started happening frequently.

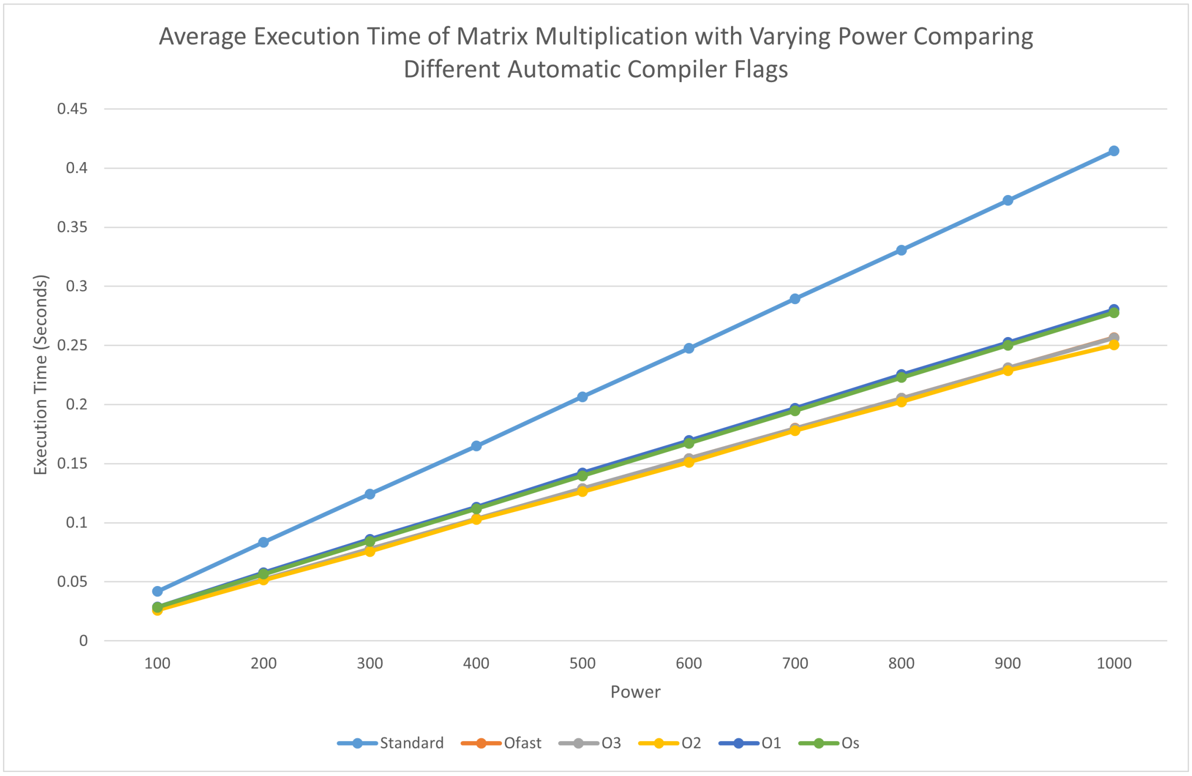

Figure 1.4: Execution times for varying matrix powers using different compiler optimization flags.

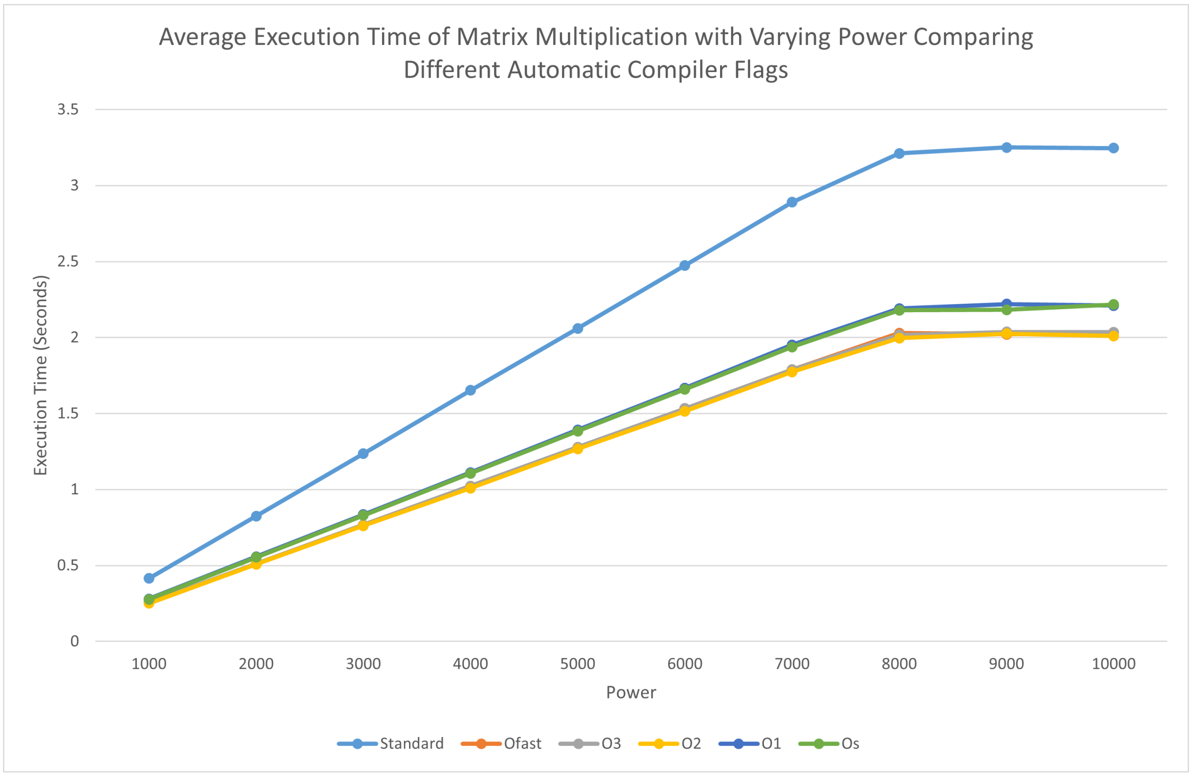

Figure 1.5: Execution times for varying matrix powers using different compiler optimization flags.

The results show that the Ofast, O2, and O3 optimization flags consistently provided the best performance, with very little time separating each of them. There was an average execution time reduction of approximately 39% for dimension scaling and 38% for power scaling compared to the standard implementation. Due to the discussion on the differences between O2 and O3 performance, for the remainder of the lab the choice was made to use the O3 compiler flag. Further manual optimizations are shown, and in each of the manual optimizations, the code is executed with the standard compile command, and the O3 flag. Then, both options are tested. This is to get a range of slowest (standard compile) and fastest (O3 optimized compile) for each of the manual improvement implementations.

1.5.3 Impact of Flattened Arrays

The next set of experiments evaluated the performance impact of flattening the 2D dynamic arrays into 1D arrays. The results demonstrated significant improvements in execution times.

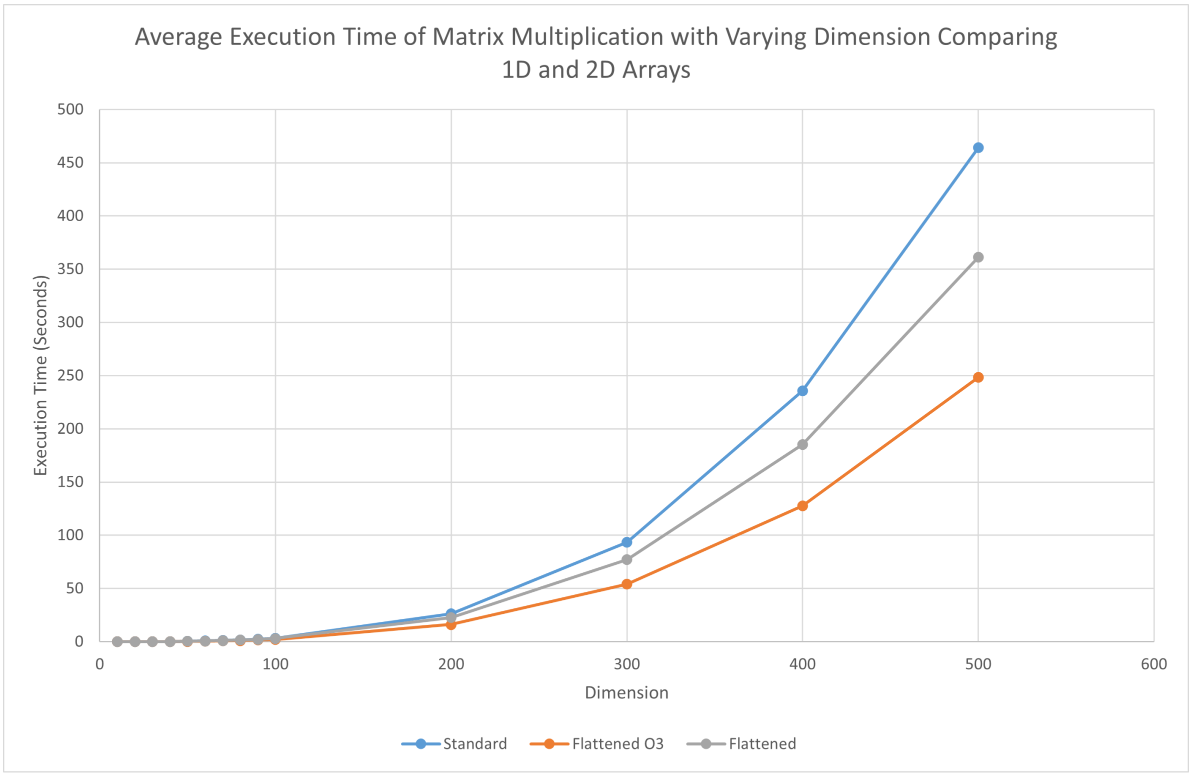

Figure 1.6: Percent improvement in execution times for varying dimensions when using flattened arrays.

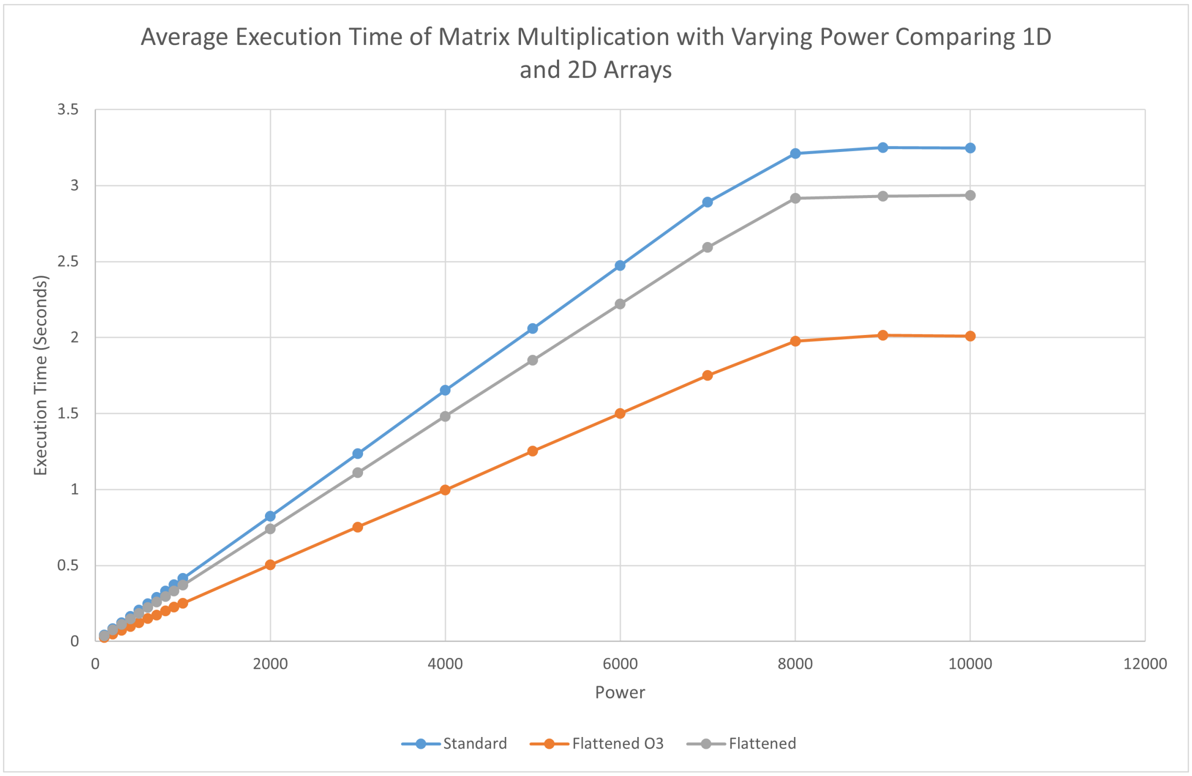

Figure 1.7: Percent improvement in execution times for varying powers when using flattened arrays.

Flattening the arrays resulted in an average execution time reduction of approximately 11% when running the basic experiments, and 39% improvement when running the O3 optimized experiments, with the most significant improvements observed for larger matrices and higher powers.

1.5.4 Register Variable Optimization

Further optimizations were made by using register variables to reduce memory access times. This approach yielded substantial performance gains.

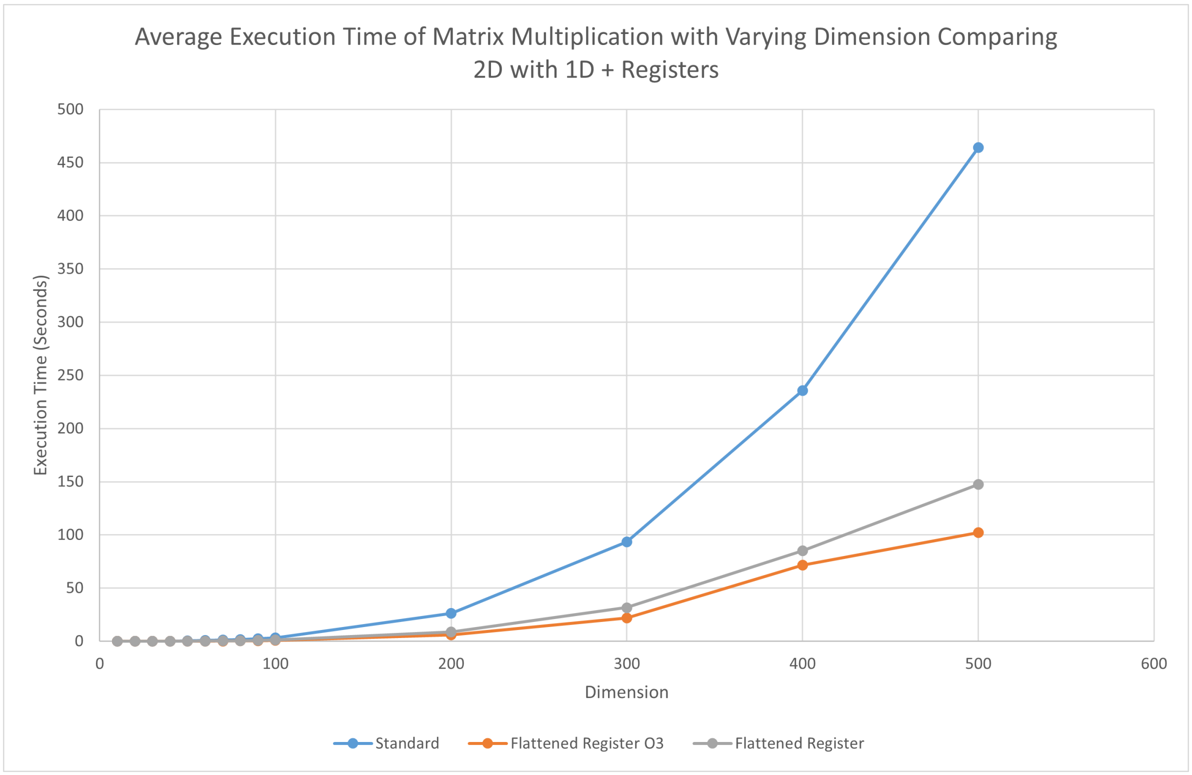

Figure 1.8: Percent improvement in execution times for varying dimensions when using flattened arrays and register variables.

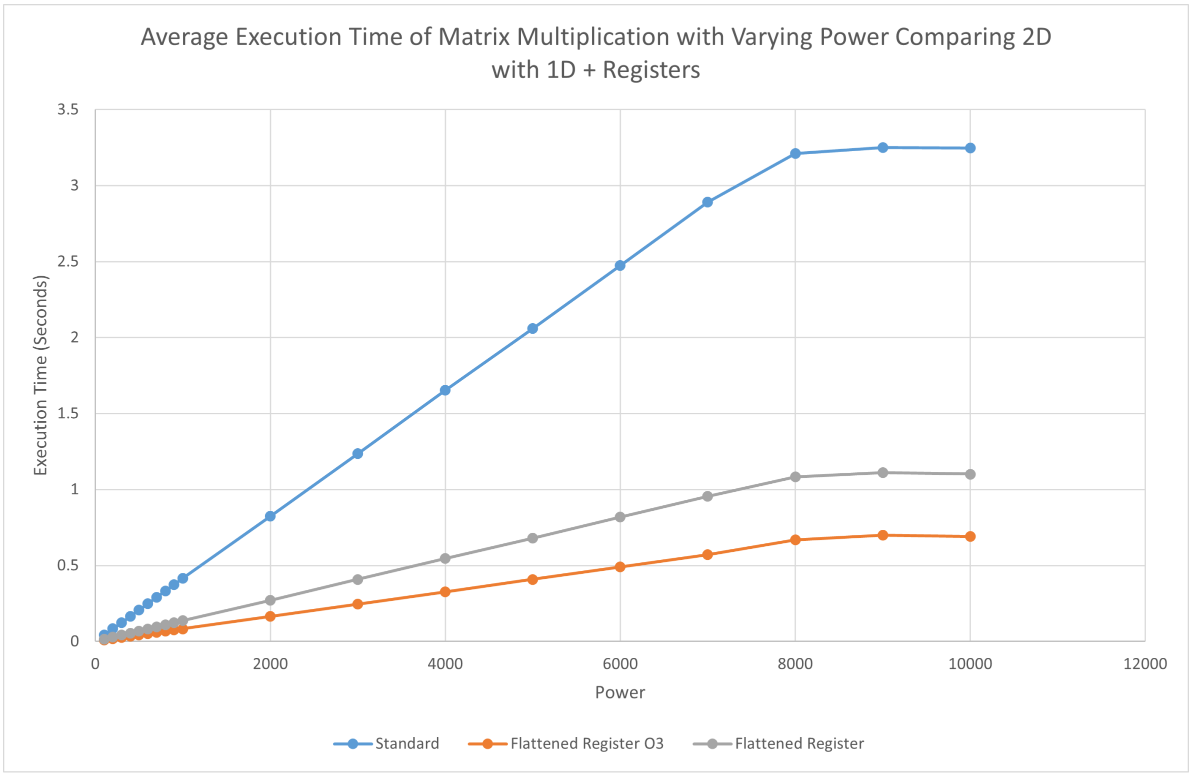

Figure 1.9: Percent improvement in execution times for varying powers when using flattened arrays and register variables.

Using register variables along with flattened arrays and the O3 compiler flag resulted in an average execution time reduction of approximately 76.76% for dimension scaling and 79.76% for power scaling.

1.5.5 Scalability Analysis

To evaluate the scalability of the optimized implementations, we analyzed the performance for both small and large matrix dimensions and powers.

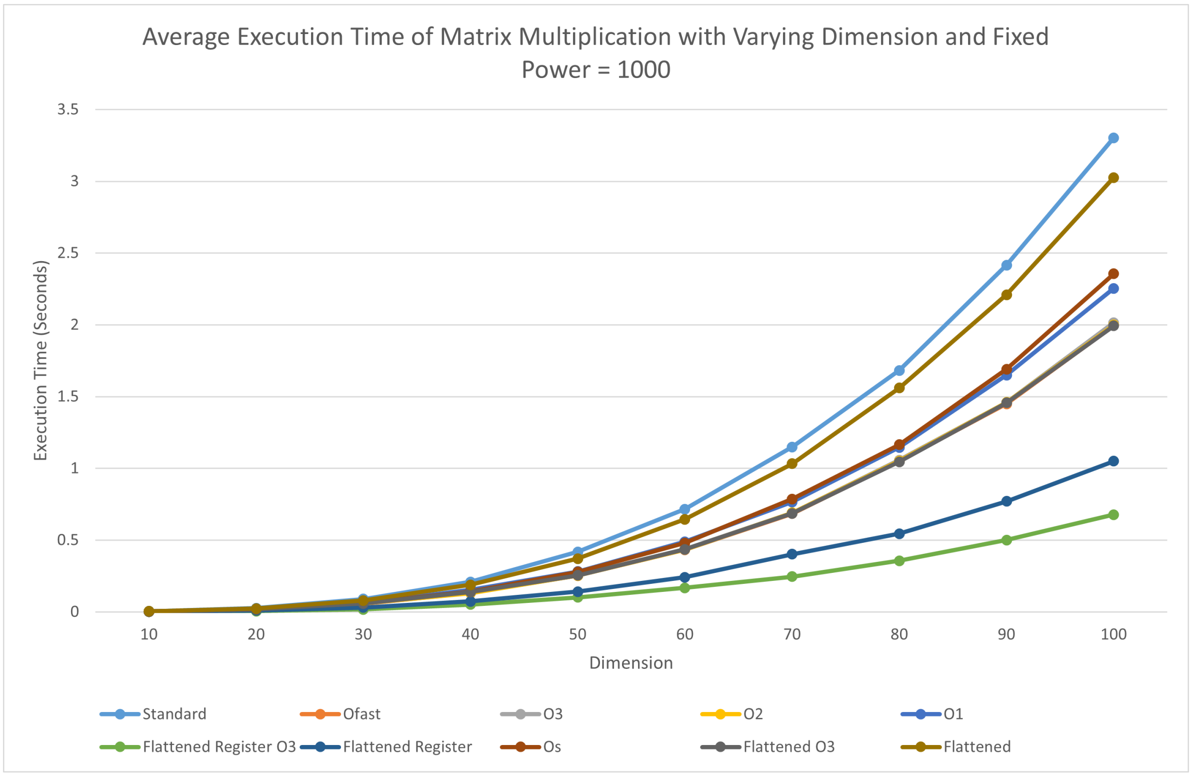

Figure 1.10: Scalability analysis for small matrix dimensions.

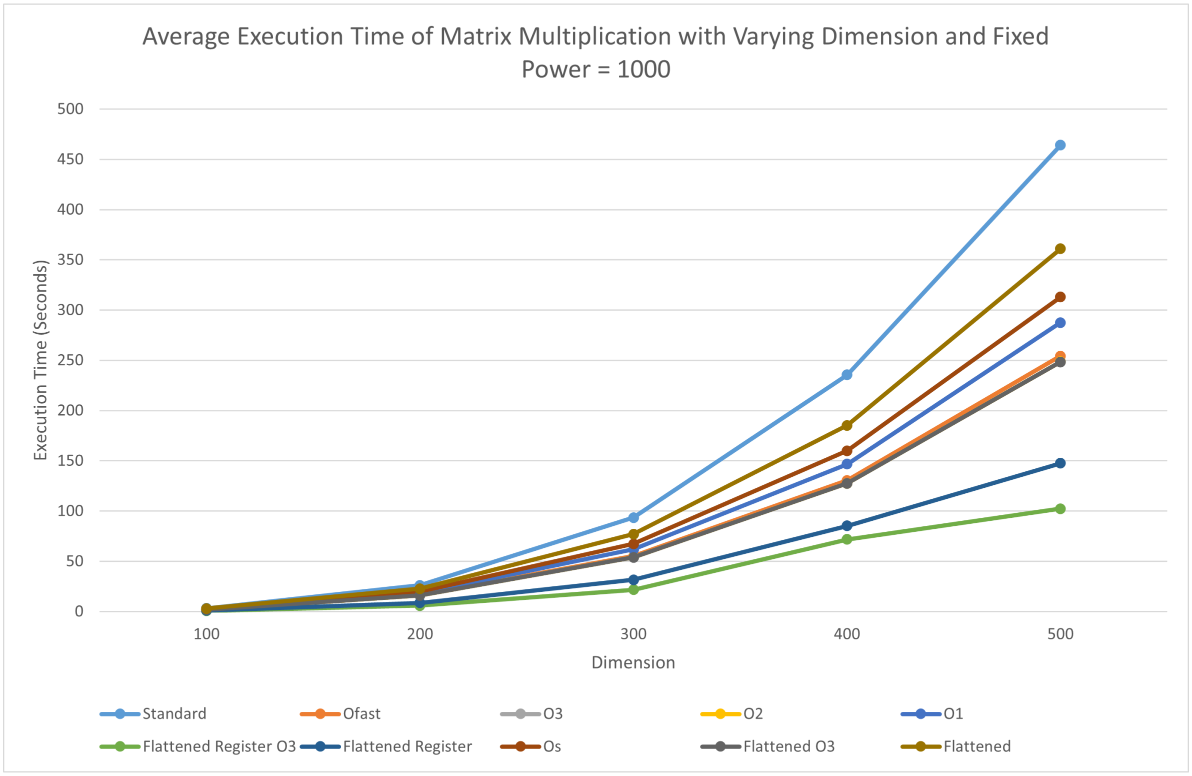

Figure 1.11: Scalability analysis for large matrix dimensions.

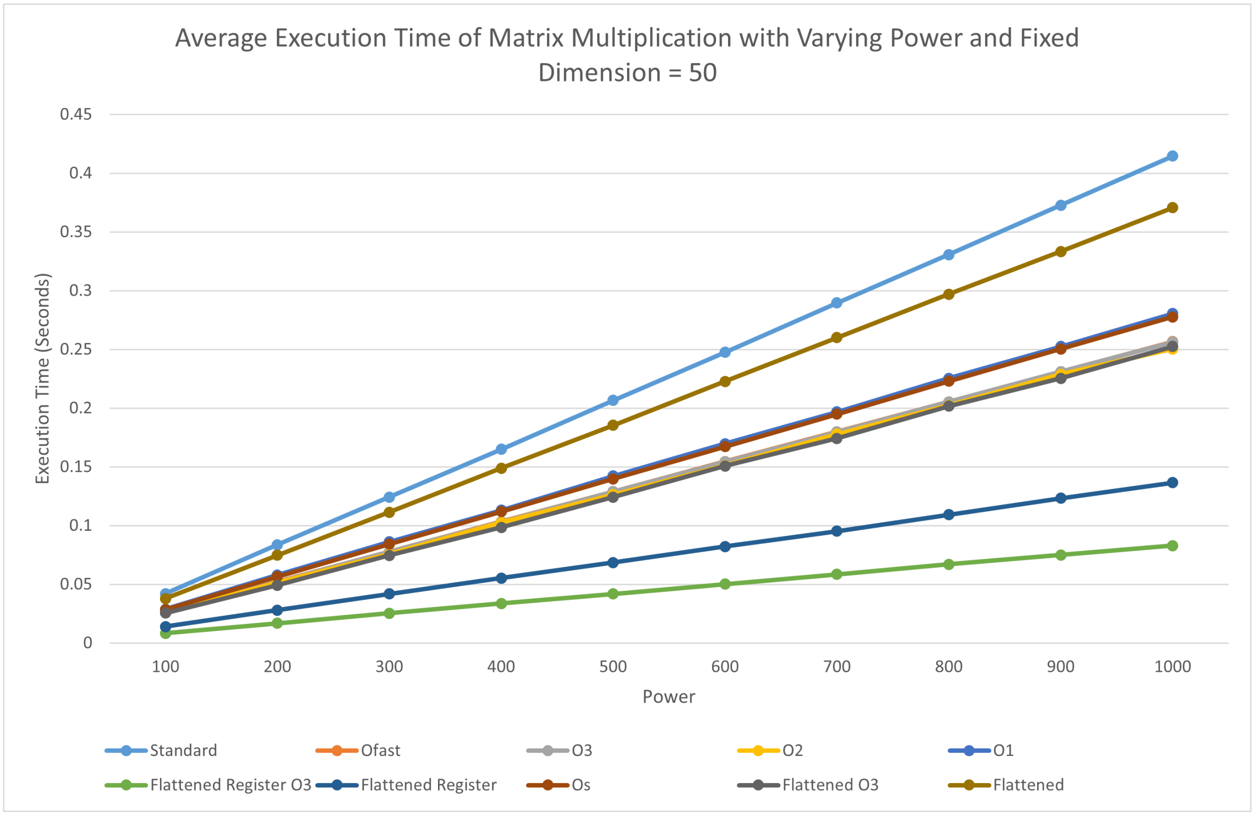

Figure 1.12: Scalability analysis for small matrix powers.

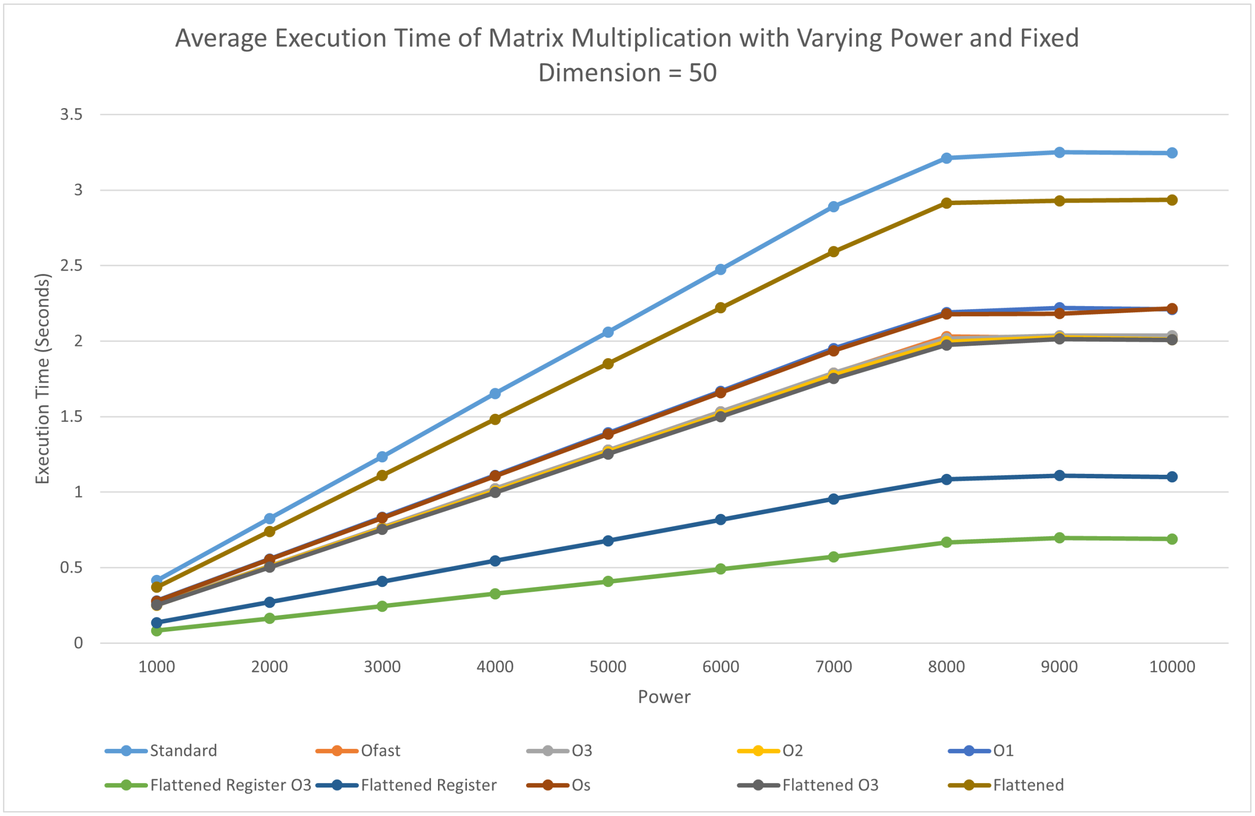

Figure 1.13: Scalability analysis for large matrix powers.

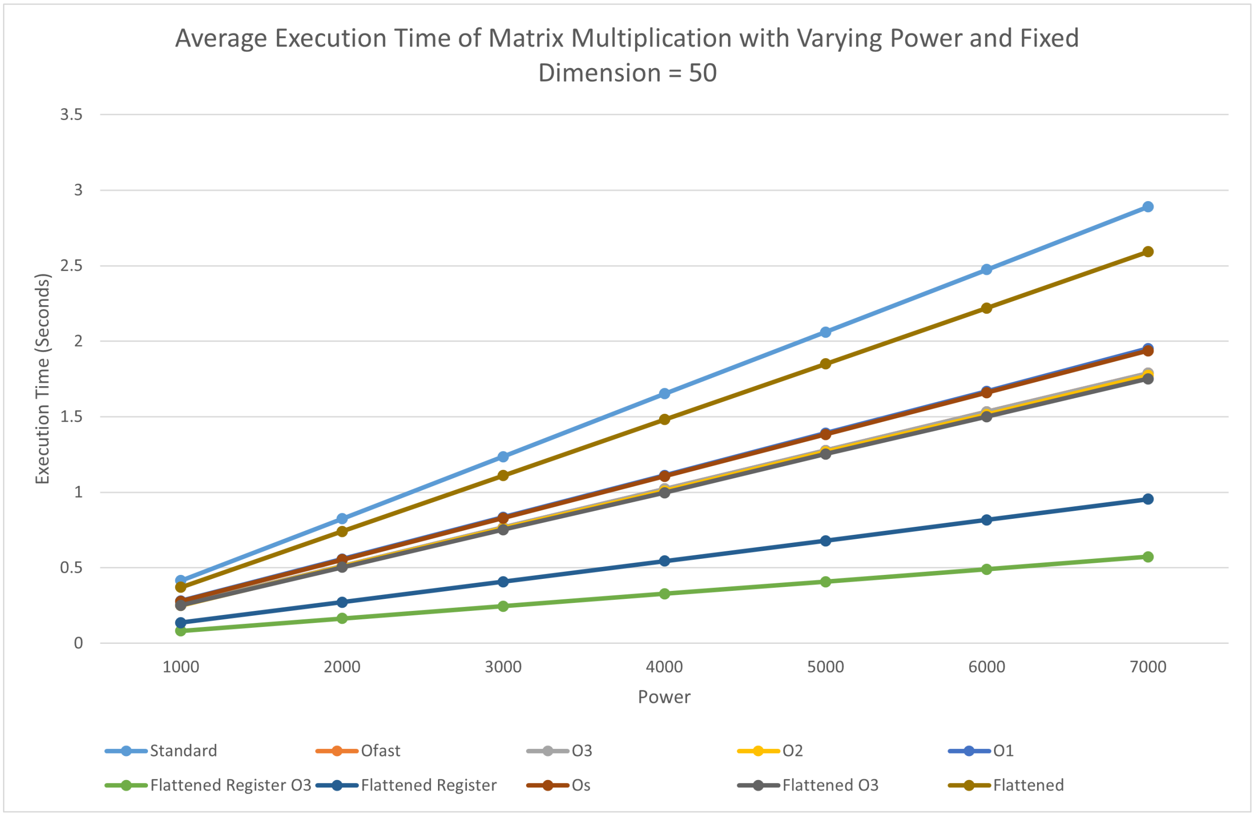

Figure 1.14: Scalability analysis for large matrix powers, capped at the point where matrices started being NaN-like.

From these figures, we can see the huge improvements simply by making a few changes from the standard 2D array implementation, to the 1D + register-based implementation alongside the O3 compile flag. The performance increase is incredible, and can further be seen in the following tables:

Dimension	Standard	O1	O2	O3	Os	Ofast	Flattened	Flattened O3	Register	Register O3
10	0.0037	0.0025	0.0023	0.0026	0.0026	0.0025	0.0035	0.0023	0.0026	0.0011
20	0.0269	0.0182	0.0178	0.0173	0.0185	0.0167	0.0238	0.0171	0.0094	0.0056
30	0.0890	0.0633	0.0585	0.0583	0.0625	0.0561	0.0797	0.0578	0.0303	0.0182
40	0.2093	0.1522	0.1305	0.1331	0.1431	0.1326	0.1885	0.1378	0.0742	0.0515
50	0.4175	0.2805	0.2534	0.2545	0.2777	0.2527	0.3708	0.2542	0.1419	0.1013
60	0.7154	0.4890	0.4344	0.4362	0.4801	0.4327	0.6454	0.4350	0.2410	0.1672
70	1.1490	0.7660	0.6900	0.6899	0.7861	0.6849	1.0325	0.6861	0.4022	0.2451
80	1.6822	1.1458	1.0522	1.0570	1.1648	1.0522	1.5598	1.0462	0.5452	0.3567
90	2.4164	1.6497	1.4601	1.4589	1.6917	1.4486	2.2093	1.4571	0.7708	0.5009
100	3.3032	2.2536	1.9994	2.0145	2.3558	1.9965	3.0263	1.9926	1.0502	0.6775
200	26.2026	18.1342	16.1940	16.1920	19.7596	16.4858	22.7694	16.1359	8.6037	5.8248
300	93.5694	61.9804	54.1024	54.0871	67.5039	55.2833	77.1486	54.0000	31.6398	21.8022
400	235.5102	146.7287	127.5814	127.5535	160.0460	130.4646	185.1545	127.5117	85.2391	71.7359
500	464.2029	287.6034	248.5509	248.5077	312.9559	254.3076	361.0629	248.4256	147.6710	102.3497

Table 1.1: Execution times (in seconds) for varying dimensions with different optimization techniques.

Power	Standard	O1	O2	O3	Os	Ofast	Flattened	Flattened O3	Register	Register O3
100	0.0420	0.0287	0.0259	0.0264	0.0284	0.0262	0.0378	0.0256	0.0139	0.0084
200	0.0836	0.0579	0.0516	0.0521	0.0565	0.0522	0.0748	0.0491	0.0279	0.0168
300	0.1243	0.0861	0.0757	0.0777	0.0843	0.0777	0.1114	0.0746	0.0418	0.0253
400	0.1649	0.1131	0.1026	0.1032	0.1118	0.1033	0.1488	0.0984	0.0552	0.0337
500	0.2065	0.1422	0.1263	0.1289	0.1398	0.1287	0.1854	0.1243	0.0685	0.0418
600	0.2475	0.1696	0.1511	0.1543	0.1673	0.1545	0.2227	0.1507	0.0821	0.0502
700	0.2895	0.1968	0.1779	0.1797	0.1948	0.1799	0.2600	0.1741	0.0953	0.0585
800	0.3307	0.2253	0.2022	0.2053	0.2229	0.2053	0.2970	0.2018	0.1093	0.0670
900	0.3728	0.2525	0.2288	0.2311	0.2503	0.2308	0.3334	0.2254	0.1233	0.0749
1000	0.4145	0.2805	0.2503	0.2563	0.2777	0.2566	0.3706	0.2524	0.1365	0.0829
2000	0.8250	0.5576	0.5071	0.5116	0.5538	0.5114	0.7404	0.5033	0.2719	0.1642
3000	1.2348	0.8349	0.7611	0.7669	0.8296	0.7669	1.1104	0.7532	0.4085	0.2457
4000	1.6530	1.1113	1.0088	1.0220	1.1062	1.0221	1.4821	0.9976	0.5449	0.3275
5000	2.0599	1.3923	1.2673	1.2776	1.3835	1.2773	1.8505	1.2528	0.6788	0.4085
6000	2.4746	1.6685	1.5155	1.5328	1.6594	1.5326	2.2202	1.5001	0.8177	0.4907
7000	2.8902	1.9513	1.7734	1.7884	1.9368	1.7879	2.5921	1.7516	0.9546	0.5722
8000	3.2113	2.1905	1.9962	2.0167	2.1797	2.0301	2.9153	1.9754	1.0838	0.6681
9000	3.2508	2.2203	2.0248	2.0367	2.1818	2.0214	2.9291	2.0142	1.1102	0.6981
10000	3.2462	2.2107	2.0094	2.0356	2.2168	2.0341	2.9354	2.0083	1.1012	0.6897

Table 1.2: Execution times (in seconds) for varying powers with different optimization techniques.

The results indicate that the optimized implementations scale well with increasing matrix dimensions and powers, demonstrating the effectiveness of the applied optimization techniques.

We can also view the percent improvement in terms of execution time for each of the various stages of optimization.

Dimension	O1	O2	O3	Os	Ofast	Flattened	Flattened O3	Register	Register O3
10	31.4934%	37.7997%	28.9542%	28.3594%	32.7415%	5.4816%	36.9549%	29.7261%	70.0248%
20	32.3865%	33.7999%	35.6451%	31.2579%	37.9580%	11.3681%	36.2773%	65.1189%	79.1167%
30	28.8580%	34.2812%	34.4714%	29.6938%	36.9461%	10.3756%	34.9749%	65.9623%	79.5954%
40	27.2832%	37.6409%	36.4173%	31.6308%	36.6677%	9.9699%	34.1782%	64.5688%	75.3944%
50	32.8202%	39.2912%	39.0499%	33.4856%	39.4589%	11.1734%	39.1023%	66.0042%	75.7327%
60	31.6512%	39.2771%	39.0234%	32.8879%	39.5194%	9.7824%	39.1866%	66.3128%	76.6250%
70	33.3276%	39.9448%	39.9517%	31.5823%	40.3885%	10.1315%	40.2854%	64.9951%	78.6672%
80	31.8860%	37.4549%	37.1680%	30.7584%	37.4522%	7.2759%	37.8090%	67.5886%	78.7974%
90	31.7302%	39.5755%	39.6262%	29.9942%	40.0510%	8.5729%	39.7022%	68.1005%	79.2722%
100	31.7761%	39.4707%	39.0134%	28.6805%	39.5581%	8.3833%	39.6763%	68.2064%	79.4881%
200	30.7922%	38.1969%	38.2045%	24.5890%	37.0834%	13.1024%	38.4188%	67.1649%	77.7701%
300	33.7600%	42.1794%	42.1957%	27.8568%	40.9173%	17.5493%	42.2894%	66.1858%	76.6995%
400	37.6975%	45.8277%	45.8395%	32.0429%	44.6034%	21.3816%	45.8572%	63.8066%	69.5402%
500	38.0436%	46.4564%	46.4657%	32.5821%	45.2163%	22.2187%	46.4834%	68.1883%	77.9515%
Average	**32.3933%**	**39.3712%**	**38.7162%**	**30.3858%**	**39.1830%**	**11.9119%**	**39.3711%**	**63.7092%**	**76.7625%**

Table 1.3: Percent improvement in execution times for varying dimensions with different optimization techniques.

Power	O1	O2	O3	Os	Ofast	Flattened	Flattened O3	Register	Register O3
100	31.6543%	38.1719%	37.1901%	32.2275%	37.5476%	9.9816%	39.0903%	66.8353%	80.0705%
200	30.8174%	38.3525%	37.6837%	32.4089%	37.6284%	10.5302%	41.2424%	66.6398%	79.8633%
300	30.7064%	39.0781%	37.4894%	32.1743%	37.4814%	10.3879%	39.9886%	66.3847%	79.6137%
400	31.3968%	37.7528%	37.4033%	32.1756%	37.3385%	9.7933%	40.3021%	66.5190%	79.5708%
500	31.1754%	38.8538%	37.5832%	32.3339%	37.6701%	10.2574%	39.8111%	66.8484%	79.7652%
600	31.4702%	38.9465%	37.6625%	32.4190%	37.5919%	10.0188%	39.1091%	66.8376%	79.7232%
700	32.0175%	38.5439%	37.9349%	32.7143%	37.8477%	10.2001%	39.8518%	67.0821%	79.8010%
800	31.8769%	38.8647%	37.9338%	32.6101%	37.9107%	10.1835%	38.9887%	66.9582%	79.7537%
900	32.2588%	38.6159%	38.0085%	32.8540%	38.0736%	10.5671%	39.5202%	66.9145%	79.9035%
1000	32.3334%	39.6159%	38.1697%	33.0131%	38.1031%	10.5986%	39.1082%	67.0608%	79.9922%
2000	32.4139%	38.5385%	37.9915%	32.8715%	38.0181%	10.2554%	39.0003%	67.0472%	80.0918%
3000	32.3815%	38.3618%	37.8913%	32.8144%	37.8908%	10.0726%	38.9983%	66.9140%	80.1019%
4000	32.7691%	38.9729%	38.1719%	33.0772%	38.1666%	10.3406%	39.6515%	67.0369%	80.1896%
5000	32.4080%	38.4772%	37.9755%	32.8338%	37.9895%	10.1674%	39.1798%	67.0447%	80.1698%
6000	32.5765%	38.7587%	38.0604%	32.9422%	38.0676%	10.2795%	39.3824%	66.9568%	80.1721%
7000	32.4845%	38.6400%	38.1224%	32.9857%	38.1372%	10.3143%	39.3957%	66.9710%	80.2035%
8000	31.7898%	37.8401%	37.2012%	32.1249%	36.7840%	9.2196%	38.4871%	66.2506%	79.1956%
9000	31.6986%	37.7146%	37.3484%	32.8821%	37.8177%	9.8945%	38.0405%	65.8494%	78.5266%
10000	31.8978%	38.0992%	37.2931%	31.7094%	37.3379%	9.5733%	38.1333%	66.0783%	78.7529%
Average	**31.9014%**	**38.5368%**	**37.7429%**	**32.5880%**	**37.7580%**	**10.1387%**	**39.3306%**	**66.7489%**	**79.7611%**

Table 1.4: Percent improvement in execution times for varying powers with different optimization techniques.

1.5.6 Discussion of Results

The experimental results highlight the significant impact of various optimization techniques on the performance of matrix exponentiation. Key findings include:

- The Ofast optimization flag provided the best performance among the compiler flags on average, reducing execution time by approximately 39.18% for dimension scaling and 37.76% for power scaling.
- Simply flattening the arrays to a 1D implementation resulted in an average execution time reduction of approximately 11.91% for dimension scaling and 10.14% for power scaling.
  - This was further sped up to an average of 39.37% for dimension scaling and 39.33% for power scaling when using the 1D implementation and the 03 compiler flag
- Using register variables, combined with flattened arrays and the O3 compiler flag, achieved an average execution time reduction of approximately 76.76% for dimension scaling and 79.76% for power scaling.
- The optimized implementations demonstrated good scalability for both small and large matrix dimensions and powers.

These results underscore the importance of efficient memory access patterns and the use of fast access storage like registers in high-performance computing. The optimization techniques applied in this study significantly improved the execution time of matrix exponentiation, making them valuable strategies for similar computational tasks.

1.6 Conclusion

This lab focused on implementing and optimizing the power operation on large square matrices using various optimization techniques in C/C++. The primary goal was to assess the performance improvements achievable through different compiler optimizations and manual code optimizations, and to gain insights into the efficiency of these methods.

1.6.1 Summary of Findings

The experiments conducted in this lab demonstrated significant performance gains through both compiler optimizations and manual code optimizations. Specifically, the use of compiler optimization flags such as `-O1`, `-O2`, `-O3`, `-Os`, and `-Ofast` resulted in substantial reductions in execution time compared to the baseline implementation. Among these, the `-O3` and `-Ofast` flags consistently provided the best performance improvements, with average speedups of approximately 38% and 39% for varying dimensions and powers, respectively.

Further performance enhancements were achieved through manual code optimizations. Converting the 2D dynamic arrays to 1D flattened arrays significantly reduced execution time by improving memory access patterns and increasing cache hits. This change alone resulted in an average speedup of about 12% for varying dimensions and 10% for varying powers. Additionally, the use of register variables in critical loops of the matrix multiplication function led to remarkable performance gains, with speedups of around 64% for varying dimensions and 67% for varying powers. When combined with the `-O3` compiler optimization flag, these manual optimizations resulted in overall speedups of approximately 77% for varying dimensions and 80% for varying powers.

1.6.2 Implications for High-Performance Computing

The findings from this lab have important implications for high-performance computing applications. Efficient memory management and the use of appropriate compiler optimizations are crucial for achieving high performance in computationally intensive tasks such as matrix operations. The substantial speedups observed in this lab highlight the potential of these techniques to significantly reduce computation time and improve the overall efficiency of HPC applications.

Moreover, the manual optimizations performed in this lab, including the use of flattened arrays and register variables, demonstrate the importance of low-level code optimizations in achieving maximum performance. These techniques can be particularly beneficial in scenarios where compiler optimizations alone are insufficient to meet performance requirements.

1.6.3 Discussion of Unexpected Results

While the performance improvements achieved in this lab were generally consistent with expectations, there were a few unexpected findings. Notably, the `-O2` and `-O3` compiler optimization flags produced very similar speedups, with `-O3` being only marginally faster in most cases. This suggests that the additional optimizations performed by `-O3` may have limited impact on certain types of code or datasets. Additionally, the extent of the performance gains from using register variables exceeded initial expectations, highlighting the significant impact of efficient variable management on computation speed.

1.6.4 Future Work

Future work in this area could involve extending the optimizations explored in this lab to other types of matrix operations, such as matrix inversion and eigenvalue computations. Additionally, implementing these optimizations in other programming languages and comparing their performance could provide valuable insights into the portability and effectiveness of these techniques.

Further investigations could also focus on optimizing matrix operations for different hardware architectures, including GPUs and distributed computing environments. Leveraging parallel processing capabilities and exploring advanced memory management techniques could lead to significant performance improvements in HPC applications.

1.6.5 Conclusion

In conclusion, this lab successfully demonstrated the significant performance benefits achievable through both compiler and manual optimizations for matrix power operations. The insights gained from this lab underscore the importance of efficient memory management, low-level code optimizations, and the use of appropriate compiler flags in high-performance computing. These findings provide a solid foundation for future explorations and optimizations in the field of computational science and engineering.