Performance Evaluation of a Single Core

Fábio Morais, FEUP
up202008052@fe.up.pt
Paços de Ferreira, Portugal

Filipe Fonseca, FEUP
up202003474@fe.up.pt
Paços de Ferreira, Portugal

Marcos Ferreira, FEUP
up201800177@fe.up.pt
Porto, Portugal

March 9, 2023

Abstract

Our study investigated the impact of memory management and algorithm optimizations, including cache optimization, on matrix multiplication performance using both C++ and Java programming languages. We conducted experiments with various matrix sizes and algorithms and analyzed the resulting data by creating graphs and writing a report. Our analysis revealed that optimizing memory management and cache usage significantly improved performance by reducing execution time. Specifically, loading data into cache in a way that could be better utilizing it yielded better results. The study findings are consistent across both programming languages, indicating broad applicability. Overall, this study provides valuable insights into improving matrix multiplication performance through memory management and algorithm optimization.

1 Introduction

This project is the first assignment for the Parallel and Distributed Computing course at FEUP, which aims to study the effect of the memory hierarchy on the processor's performance when accessing large amounts of data. In this study, we investigated the impact of memory management and algorithm optimizations on matrix multiplication performance using C++ and Java. PAPI provided detailed metrics such as L1/L2 cache misses, total instructions per run, total cycles executed, execution time, and gigaflops per run. We implemented three different matrix multiplication algorithms in both programming languages to recognize that performance tendencies are not exclusive to C++ programs. To simulate substantial memory usage, we multiplied matrices of various sizes and analyzed the data through graphs and report writing. Our analysis focused on the effectiveness of memory management, including cache optimization, and algorithm optimizations on improving matrix multiplication performance. The study provides valuable insights into improving matrix multiplication performance through memory management and algorithm optimization, applicable to both C++ and Java programming languages.

2 Algorithms' Explanations

As part of the project, three algorithms were utilized by our team, each with its own computational complexity. Below are their details:

2.1 Basic Multiplication

The basic matrix multiplication algorithm employs the mathematical definition of matrix multiplication and utilizes three nested loops. Its time complexity is $O(N^3)$, where N is the dimension of the matrices involved.

```
for LineCount ← 0 to NumberOfLines do
    for ColumnCount ← 0 to NumberOfColumns do
        for ElementCount ← 0 to NumberOfElements do
            matrix_c[MatrixLine][MatrixColumn] ← matrixResult + Element * Column;
        end
    end
end
```

2.2 Line Multiplication

The line multiplication algorithm is a variant of the basic matrix multiplication algorithm that optimizes memory usage by rearranging the for loops to perform the multiplication of each line of the first matrix with the corresponding line of the

second matrix. This technique enables memory to be loaded into the cache more efficiently, unlike the original algorithm that multiplies elements of the matrices row by column.

```
for LineCount ← 0 to NumberOfLines do
    for ElementCount ← 0 to NumberOfElements do
        for ColumnCount ← 0 to NumberOfColumns do
            matrix_c[MatrixLine][MatrixColumn] ← matrixResult + Element ∗ Line;
        end
    end
end
```

2.3 Block Multiplication

The block-oriented multiplication algorithm divides the input matrices into smaller submatrices or blocks and performs matrix multiplication on these blocks instead of on the entire matrices. This approach is designed to take advantage of the line multiplication technique used in the previous algorithm to calculate all the values in a block. By breaking the matrices down into smaller, more manageable blocks, this algorithm can reduce the amount of data that needs to be loaded into the cache and improve memory usage.

```
for BlockLineCount ← 0 to NumberOfBlocks do
    for BlockElement ← 0 to NumberOfBlocks do
        for BlockColumnCount ← 0 to NumberOfBlocks do
            NextLineBlock ← (BlockLineCount + 1) ∗ BlockSize;
            for MatrixLine ← BlockLineCount ∗ BlockSize to NextLineBlock do
                NextCountBlock ← (BlockElement + 1) ∗ BlockSize;
                for ElementCount ← BlockElement ∗ BlockSize to NextCountBlock do
                    NextColumnBlock ← (BlockColumnCount + 1) ∗ BlockSize;
                    for MatrixColumn ← BlockColumnCount ∗ BlockSize to NextColumnBlock do
                        matrix_c[MatrixLine][MatrixColumn] ← matrix_c[MatrixLine][MatrixColumn] +
                          matrix_a[MatrixLine][MatrixColumn] ∗ matrix_b[MatrixLine][MatrixColumn];
                    end
                end
            end
        end
    end
end
```

3 Performance Metrics

In order to measure the performance of the processor, and different available measurements, we compared results in two different programming languages (Java and C++) and different. In order to measure the performance of the processor, we compared results using two different programming languages (Java and C++).

In order to ensure greater precision in our data, we conducted each test 10 times and examined the varying outcomes. Ideally, we would have preferred to conduct at least 30 runs per test, but due to time constraints, this was not possible. The objective behind this approach was to verify the consistency of the data and attenuate the impact of external factors on the results.

3.1 Setup

Our group utilized information for the data cache miss (DCM) registers for both L1 and L2 that were read directly for/ supported by the hardware. These registers are crucial indicators of the effectiveness of cache usage and the impact it has on the execution time of the code.

We also considered it important to measure the total number of instructions executed (TOT_INS) and total number of cycles elapsed (TOT_CYC) to calculate the cycles per instruction (CPI) as it provides a good indication of the processor's performance.

In addition, we attempted to measure L1 instruction cache misses (ICM), L1 data cache misses (DCM), L1 cache store misses (STM), and full core cycles (FUL CCY) to further develop the project. However, this was not feasible due to

incompatibility with our hardware/program. When we tried to execute the program, we encountered an error at the start of the execution, and the register would be filled with a random value, making it difficult to accurately measure the desired metrics.

In order to streamline the data gathering process, we developed a bash script that enabled us to efficiently collect and organize the data into CSV files. Subsequently, the collected data was processed using Python, along with libraries such as Pandas, NumPy, and Matplotlib. The Python scripts computed the minimum, maximum, average, and standard deviation values, which were utilized in generating the graphs and tables presented below. In order to achieve automation, we made necessary changes to the code, including the incorporation of "human_prints" that eliminated the need for a run loop and produced suitable output for conversion into a CSV file.

Specs:
CPU: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz.
L1 Data Cache: 32 KB
L1 Instruction Cache: 32 KB
L2 Unified Cache: 256 KB
L3 Unified Cache: 12288 KB

4 Data

After conducting the experiments, we have gathered a significant amount of data that will be essential for our analysis. This data encompasses various parameters and variables that were measured during the experiments and will help us draw meaningful conclusions.

Here is the data for the first experiment:

Language	Matrix Size	Time	Time STD	GigaFlops	GigaFlop STD	L1 DCM	L1 DCM STD	L2 DCM	L2 DCM STD	Cycles per Instruction (CPI)
C++	600	0.2138892	0.00718674688885343	2.02168070198657	0.0643049988482567	244758639.3	60884.8848931766	39563094.4	848148.208654098	0.575546293521027
C++	1000	1.258494	0.0855365403659616	1.59566655531816	0.105891021638671	1224030858.4	6974385.96215179	294065783	23623526.4373273	0.749133178650777
C++	1400	3.675294	0.114279986602107	1.49449683208904	0.0458623417808879	3487917835.9	44177877.372358	1515101663.4	167649046.8199	0.801759778726539
C++	1800	19.45925	0.110819264771268	0.599423885298229	0.00340142069474741	9084970282	6079274.02350225	8372374107.1	400099248.332585	2.00878924963051
C++	2200	41.03099	0.286808342703704	0.519045075842379	0.00361582158446411	17636290380.3	11163082.047531	23278000899.3	725226206.608013	2.32093031486157
C++	2600	74.0307	0.444225443265517	0.474845438073681	0.00285188037444914	30894927750.9	12893682.2598305	52283206171.1	639246520.231035	2.53888651991204
C++	3000	123.699	0.818417307300432	0.436560739327772	0.00288758698212358	50296996157.2	7530520.77287804	97365332920.8	1063070160.63092	2.76201772849001
Java	600	0.23349477	0.0112737269682814	1.85376465487622	0.0832956218011442	not applicable	not applicable	not applicable	not applicable	not applicable
Java	1000	1.691882259	0.0249864174890848	1.18234792276999	0.017498407658406	not applicable	not applicable	not applicable	not applicable	not applicable
Java	1400	4.7151583472	0.0444602090643537	1.16399841060376	0.0109167926813589	not applicable	not applicable	not applicable	not applicable	not applicable
Java	1800	20.0304679866	0.482572302546972	0.582608528713012	0.0136377471287939	not applicable	not applicable	not applicable	not applicable	not applicable
Java	2200	42.3849015836	0.317012819302387	0.502468523495803	0.00378175113563464	not applicable	not applicable	not applicable	not applicable	not applicable
Java	2600	75.2700007471	0.683585900792039	0.467046241344835	0.00417907871370997	not applicable	not applicable	not applicable	not applicable	not applicable
Java	3000	123.3114342903	0.905223562345208	0.43793677045784	0.00320530406991458	not applicable	not applicable	not applicable	not applicable	not applicable

Here is the data for the third experiment:

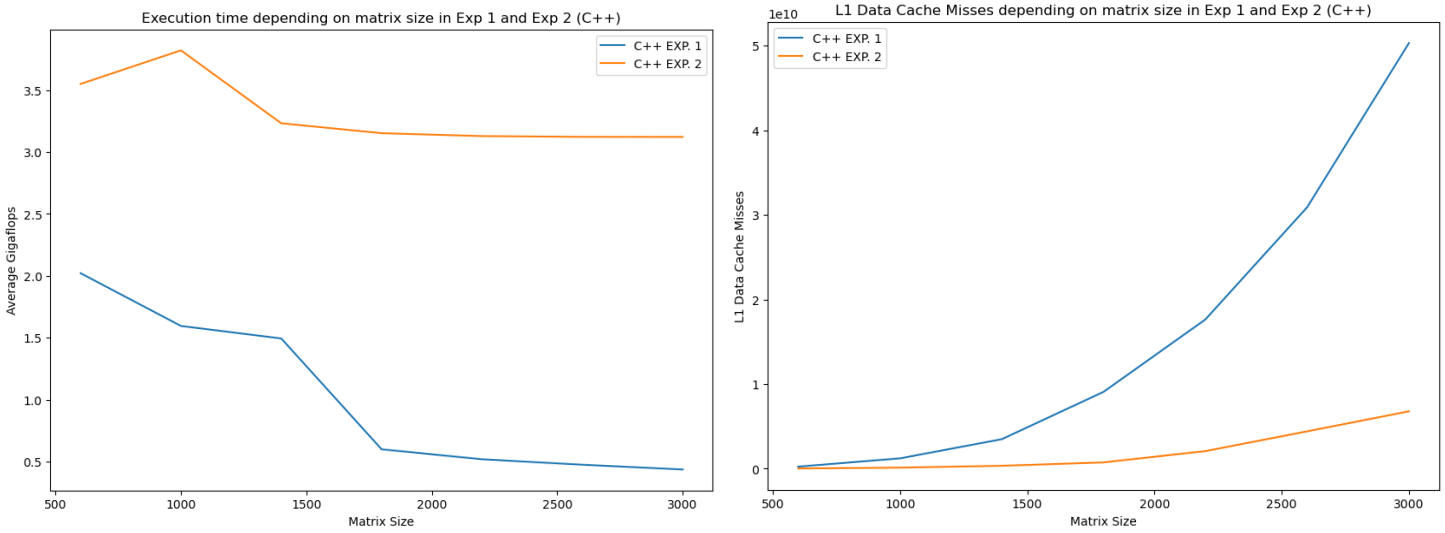
Language	Matrix Size	Time	Time STD	GigaFlops	GigaFlop STD	L1 DCM	L1 DCM STD	L2 DCM	L2 DCM STD	Cycles per Instruction (CPI)
C++	600	0.1233141	0.0169996142861471	3.55001873610571	0.378796973455794	27121496.6	14172.9873052772	56109039.3	1133778.53859395	0.289756223980432
C++	1000	0.5234591	0.00549042557447858	3.82111707920486	0.040173694806727	125692075.6	32962.0148258242	258848054.4	4374618.5261268	0.275262802465081
C++	1400	1.698113	0.0213098960057945	3.23227886368929	0.0403972878864124	346189668.1	177676.43181697	700356083.5	6618095.4833663	0.325399582849386
C++	1800	3.700324	0.0331144963892117	3.15238277261795	0.0281070031279217	746606396.4	1163338.46629355	1480698570.3	17132942.604876	0.334170783078604
C++	2200	6.806404	0.040489980708538	3.12891755560272	0.0185772244632962	2078892139	1442271.97159166	2678311245.2	35301038.8662311	0.337267037908377
C++	2600	11.25703	0.0369515012475061	3.12270113002889	0.0102327131178635	4413648406.9	49112.3619909187	4371258583.4	47810679.5590507	0.338210822850271
C++	3000	17.29631	0.0621511142297557	3.12208941312288	0.0111974460174624	6780880566.6	72947.5378842602	6760541930.6	109677177.878567	0.338822782236572
C++	4096	44.17577	0.955253453336396	3.11243089509401	0.064069785066542	17673501307.5	4742344.17321055	17215587827.1	216018748.878106	0.339714896535829
C++	6144	148.2215	0.899041248590227	3.12958488772465	0.0189073072448358	59621656294.7	18299727.0963195	58704106458.7	744734989.654918	0.338384119177038
C++	8192	351.9112	1.26707000595862	3.12443721875681	0.0112449809562415	141169305677.9	52180819.4638961	142382339869	1975806402.67177	0.339344121566146
C++	10240	692.9678	3.84417385431799	3.09905130826216	0.0170879467673693	275525886858.7	58757216.7330238	297776156482.7	3747137855.97239	0.3418840728061
Java	600	0.1700321166	0.0107316464158612	2.54895341795235	0.145627362529375	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
Java	1000	0.7214757164	0.00705764171455232	2.77233293079078	0.0269050062196411	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
Java	1400	2.2252349102	0.024696083720221	2.46653018705967	0.0274042832420508	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
Java	1800	4.7860676948	0.0180186329771155	2.43710491978618	0.00919583901258387	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
Java	2200	8.7276847931	0.0860080943112168	2.44026978756683	0.0246171702078551	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
Java	2600	13.8324628431	0.232180400522121	2.54191718950603	0.0429509605309752	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
Java	3000	20.8095549918	0.0386655101905363	2.59496986030479	0.0048280256706659	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable

Finally, here is the data for the third experiment:

Language	Matrix Size	Block Size	Time	Time STD	GigaFlops	GigaFlop STD	L1 DCM	L1 DCM STD	L2 DCM	L2 DCM STD	Cycles per Instruction (CPI)
C++	4096	128	32.50278	0.353349785151577	4.22897628083472	0.0456983157172531	9711219168.5	10321065.2931253	31804613226.9	759823838.157162	0.291780406960921
C++	4096	256	28.42181	0.175919744643845	4.83585290768083	0.0299322502404874	9072454220.7	667935.561696702	22339857605.1	336170380.898451	0.256670534999335
C++	4096	512	28.25837	0.222990194353423	4.86392810835304	0.0385024150156945	8756360237.7	1148892.68265982	19164211790.7	236313204.176345	0.25619665837482
C++	6144	128	110.0042	0.642808905421269	4.21684547986511	0.0246239311467698	32754072520.5	35545466.1493071	107123025699.2	2210457720.18775	0.293244683749058
C++	6144	256	96.11262	0.644611451961569	4.82637139381117	0.0322882868794096	30611710876.8	9574949.36444401	75244721648.1	1033380055.83953	0.257136865896693
C++	6144	512	97.99559	0.719816677973555	4.73367478925381	0.0351734954458208	29599201744.9	4291197.81038994	65392770926.7	1115289768.98902	0.263170499837145
C++	8192	128	376.0876	77.1196429858611	3.03407376839826	0.600529635620148	77832927485.7	304744235.318243	252851159538.5	3244754683.45428	0.424524931386028
C++	8192	256	412.1283	2.71234425256906	2.66799128010653	0.017649245984754	72942754719.5	52911826.6097653	166945758121.5	647744514.523713	0.468586484342064
C++	8192	512	322.4094	2.4967459800486	3.41048095657533	0.0264756625741032	70289328046	19732407.5106393	146017892006.9	1302192018.46932	0.367167957281774
C++	10240	128	519.065	2.72595320739164	4.13731813787415	0.0217627272870011	151281511375.8	546289761.483649	497608080275.6	9180138463.51764	0.298593625671516
C++	10240	256	452.6899	2.06167102936753	4.74391831633137	0.0216980448597677	141585911404	117720832.433862	352914982106.9	4875315288.30332	0.261290230770343
C++	10240	512	451.4372	7.70282724141668	4.75825503874841	0.0821850470699796	136936892436.5	58102358.1189829	302320669412	5124964663.09578	0.262116443932371

5 Results

5.1 C++: Basic multiplication compared to Line Multiplication



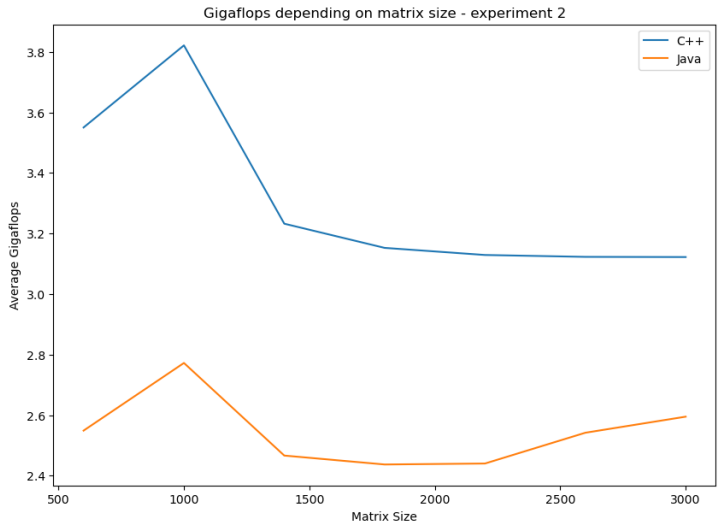
Matrix multiplication by line, also known as row multiplication, is an alternative approach that operates on a single row of the first matrix and a single row of the second matrix at a time. This approach can lead to fewer cache misses, fewer instructions needed, and faster execution times. An explanation for this behavior is because when loading a value into the cache some of its neighbors are also loaded. This means that if we need to use data next to the value requested, we already have it loaded and do not need to fetch it again. By using this technique, we can reduce the frequency of data loading and allow the processor to focus on executing the given task, instead of executing load instructions.

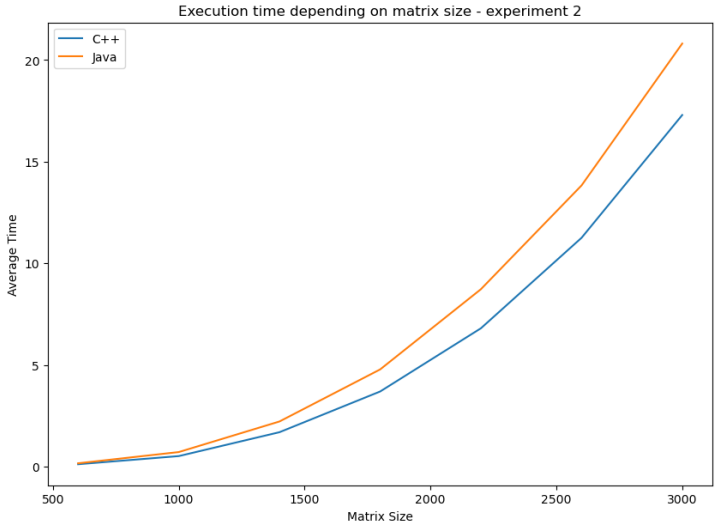
In fact, we can observe this behavior on the graph where the amount of L1 data cache misses in the first experiment is marginally bigger than the ones in the second experience, and the difference accentuates when the matrix size increases.

To better explain, caches work by loading not only the requested data but also some neighboring data into the cache, known as "cache line loading" or "cache line prefetching". This is because neighboring data is likely to be accessed soon, so it makes sense to fetch it in advance to reduce the number of cache misses. Reducing the number of cache misses is crucial because accessing data from the main memory is much slower than accessing data from the cache. By minimizing the number of cache misses, we can improve the performance of the algorithm and allow the processor to spend more time executing the actual computation, resulting in faster processing times.

5.2 Comparison between C++ and Java using Line multiplication

Time: The use of both programming languages allowed us to compare the time performance between them.



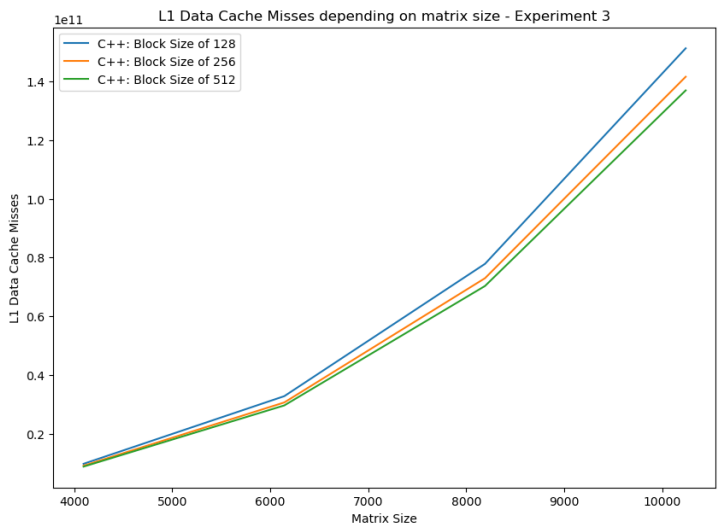


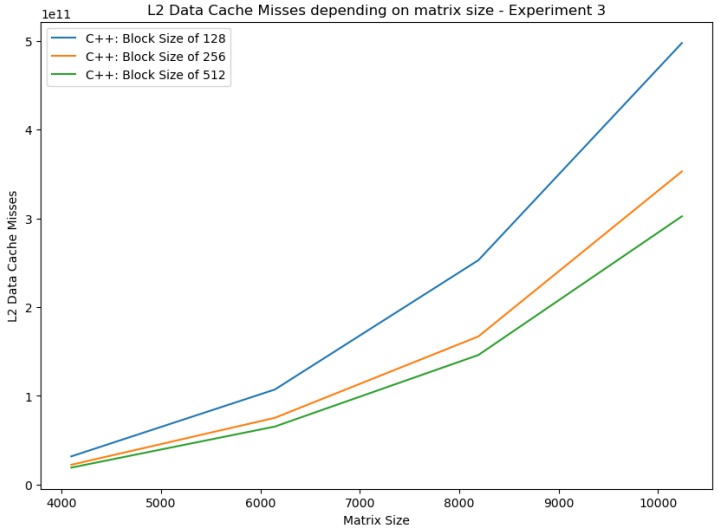
Based on the graphics presented, it is evident that C++ outperforms Java in terms of speed when executing certain types of operations. This is due to several factors, such as the differences in how C++ and Java code are compiled and executed. C++ code is compiled directly to machine code, whereas Java code is compiled to bytecode and then executed on a virtual machine (JVM). The overhead of interpreting bytecode at runtime can impact the performance of Java programs, while C++ code executes directly on the hardware, resulting in faster execution times.

One of the significant factors that contribute to C++'s faster performance is its ability to provide developers with direct access to hardware resources and system-level programming, which allows developers to fine-tune their programs for specific hardware configurations and optimize performance by minimizing overhead. Additionally, C++ allows developers to manage cache memory directly, giving them greater control over cache usage and minimizing cache misses. Cache misses can significantly affect program performance, as fetching data from main memory is much slower than accessing data in the cache. In contrast, Java does not offer the same level of control over cache management, which can result in less efficient cache usage and a higher rate of cache misses.

Additionally, based on the data obtained in the first experiment, it can be observed that C++ and Java had similar performance values, indicating that the benefits of using a more efficient language and having greater control over code behavior can be lost without proper optimization. However, in the second experiment, the performance difference between C++ and Java became more evident, particularly as the size of the matrices increased. This highlights the importance of optimizing code to fully leverage the advantages of a high-performance language like C++.

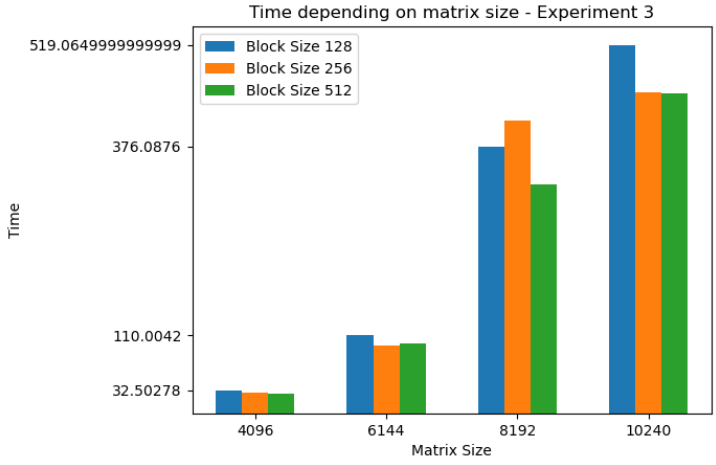
5.3 Differently sized Block multiplication

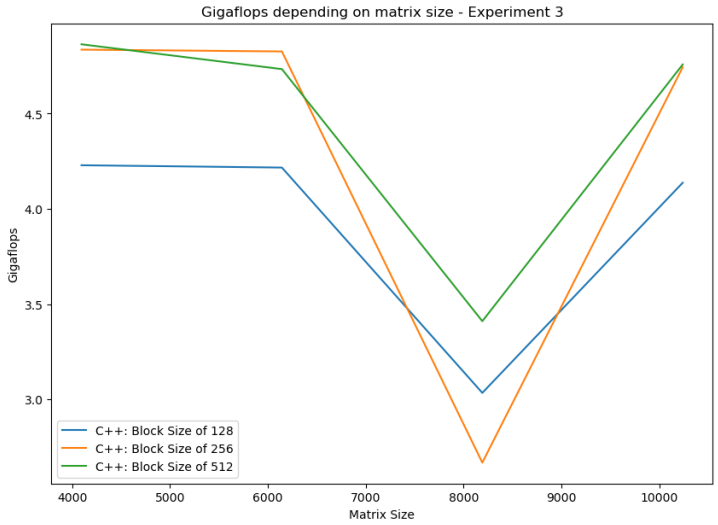




As seen from the graphs and data presented above, dividing matrices into smaller blocks can significantly improve the performance of matrix multiplication. Memory blocks can reduce the number of memory calls, resulting in faster performance.

In our configuration, we found that modern processors use a cache hierarchy to reduce the time it takes to access data from memory. Choosing a block size that fits in the smallest and fastest cache (L1 cache) can ensure that the data for each block is stored in the cache, avoiding the latency of fetching data from memory. However, a block size that is too large may not fit entirely in the L1 cache, resulting in cache misses and performance degradation. The performance difference between block sizes of 128x128 and 256x256 is significant, while the performance jump between 256x256 and 512x512 is smaller, as the 256x256 block size already fits well within the L1 cache for most modern processors. Optimizing the block size for matrix multiplication is essential for improving performance and by dividing matrices into smaller blocks and choosing a block size that fits well within the L1 cache, we can reduce cache misses and improve performance for larger amounts of data.





The higher performing configurations were found to have greater measurements of floating point operations per second (FLOPS), as can be observed in the graphs above. Conversely, poorly performing algorithms displayed decreased FLOPS, which may be attributed to inefficient memory usage. This results in the processor having to execute instructions to fetch and load data, which are slower instructions, instead of executing program instructions, negatively impacting the FLOPS scores.

Just by comparing block multiplication with differently sized blocks, we can observe a decrease in L1 and L2 data cache misses and an increase in instructions per cycle, as demonstrated in Table 3 from the data section. This can be explained by the lower number of load and store operations required. With the exception of the block size of 256 with the matrix size of 8192 that presented lower performance (which may have happened by outside factors, such as the computer performing unrelated tasks) this behavior can be observed throughout the experiments.

Overall, requiring fewer load and store operations to fetch data allows the CPU to focus more on executing the given task, resulting in better performance in useful work.

6 Conclusion

Based on the findings presented in this report, it is clear that memory management plays a crucial role in the performance of computer programs. The effective use of caches and reducing the frequency of data loading can significantly improve the execution time of algorithms. Furthermore, algorithm optimization is equally important for improving program performance. By analyzing and optimizing the use of memory, we can create more efficient programs that yield better results. These insights are not limited to matrix multiplication but are applicable to a wide range of programs. Therefore, its important to pay close attention to memory management and algorithm optimization when writing code to achieve better performance.