Faculdade de Engenharia da Universidade do Porto

CPD project 1

Performance Evaluation of a Single and Multi-core

Computação Paralela e Distribuída, 2023/24

Licenciatura em Engenharia Informática e Computação

Grupo 15 - Turma 01

Carlos Madaleno up201604906

Diana Martins up202108815

Marcos Costa up202108869

Problem Description

This assignment aimed to investigate the performance of matrix multiplication algorithms on both single-core and multi-core systems, focusing on the impact of memory hierarchy and parallelization strategies.

The project was structured into two distinct parts:

- **Part 1: Performance Evaluation of a Single Core**: we explored the effect of memory hierarchy on processor performance while accessing large amounts of data through matrix multiplication. The performance of matrix multiplication algorithms was compared across different programming languages (C/C++ and Java) to help understand the trade-offs involved in language selection for performance-critical applications.

- **Part 2: Performance Evaluation of a Multi-Core Implementation**: we parallelized the second matrix multiplication algorithm to exploit multi-core architectures and evaluate the performance of the following two solutions: 1) applying parallelization to the outermost loop and 2) executing the outermost loop in parallel by multiple threads, and parallelizing the innermost loop within each thread.

Algorithms Explanation

- **OnMult Algorithm (Naive Matrix Multiplication)**

A basic algorithm that performs matrix multiplication between two matrices. It consists of three nested loops iterating over the rows and columns of the input matrices and each element of the resulting matrix is computed by summing the products of corresponding elements from the input matrices. Initially written in C/C++, we translated it into Java, ensuring language consistency.

- **OnMultLine Algorithm (Line by Line Matrix Multiplication)**

In this algorithm, the multiplication is still done element-wise, but the order of operations is adjusted. Instead of iterating through the elements of the result matrix in row-major or column-major order, it computes each element of the result matrix by summing the products of corresponding elements from the input matrices along the same row. Similarly to what was done in the first algorithm, we then translated it into Java.

- **OnMultBlock Algorithm (Block Matrix Multiplication)**

This algorithm enhances matrix multiplication efficiency by partitioning the matrices into smaller blocks and performing multiplication block by block. It introduces additional loop structures to handle block-wise multiplication and summation.

System Specification

Our experiments were conducted on a system featuring the following specifications:

- **Processor**: Intel i7-8550U 1.80GHz;
- **L1 Cache**: 256 KB;
- **L2 Cache**: 1.0 MB;
- **L3 Cache**: 8.0 MB;
- **RAM**: 8 GB.

Part 1: Single Core Performance

Programming Language Comparison

In Figure 1, it's evident that C++ outperforms Java in calculating the product of matrices, given the same algorithm. This difference can be attributed to C++'s lower-level nature, which provides more direct control over memory management, particularly advantageous when handling large matrices. Despite this distinction, both languages exhibit similar performance trends as matrix sizes increase and when using the same algorithm. Notably, both languages benefit from the improved cache utilization offered by the OnMultLine algorithm, resulting in shorter calculation times compared to OnMult, especially with larger matrix sizes.

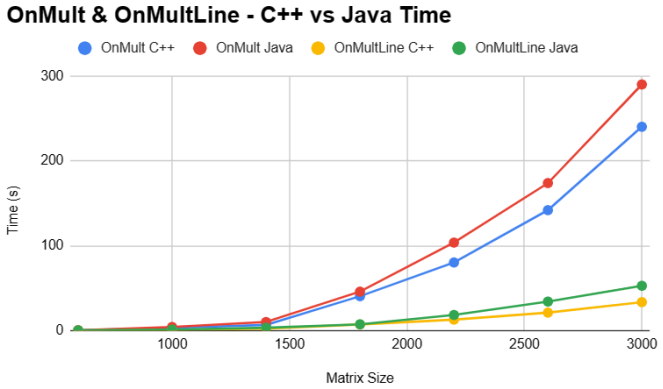

Figure 1: OnMult and OnMultLine Programming Language Time Comparison

OnMult vs OnMultLine Analysis

The analysis indicates an exponential correlation between matrix size and execution time, signifying that the rate of execution time growth exceeds that of the matrix size. Both OnMult and OnMultLine exhibit this relationship, but OnMultLine consistently outperforms OnMult across different matrix sizes due to better cache utilization. In other words, OnMultLine demonstrates a more linear growth pattern in execution time, indicating more predictable performance, and better efficiency and scalability.

OnMultLine shows fewer cache misses compared to OnMult, suggesting better cache utilization which contributes to improved performance. This is confirmed by the optimization made by switching the order of the loops, which enhances cache-friendliness by promoting sequential access to matrix elements, consequently reducing the occurrence of cache misses.

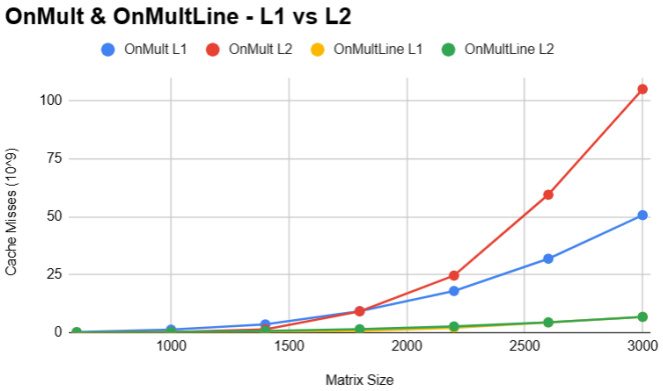

Figure 2: OnMult and OnMultLine Cache Misses

Cache misses both in L1 and L2 indicate a slower access time to memory, because L1 cache is faster than L2 cache, and both are faster than accessing data directly from RAM. In other words, for the same matrix sizes, the algorithm with the least cache misses is, in theory, faster. Optimizations aimed at improving cache locality, data reuse, and prefetching could help reduce cache misses at both cache levels and improve overall performance.

OnMultLine vs OnMultBlock Analysis

OnMultBlock can be more efficient than line-by-line multiplication for larger matrices. By working with blocks that fit into the cache, it can reduce the number of cache misses, which can significantly speed up the multiplication. In contrast, the performance of OnMultBlock depends on many factors, including the size of the matrices, the size of the blocks, and the size of the cache.

The block multiplication method is designed to take advantage of both spatial and temporal locality. However, if the block size is not a good match for the size of the matrices or the cache, this can lead to worse performance. The overhead of dividing the matrices into blocks and managing these blocks can increase the total execution time. If the block multiplication method changes the order in which elements are accessed it can lead to different cache behavior compared to the line-by-line method. In other words, blocks that are not contiguous in memory can lead to a higher number of cache misses.

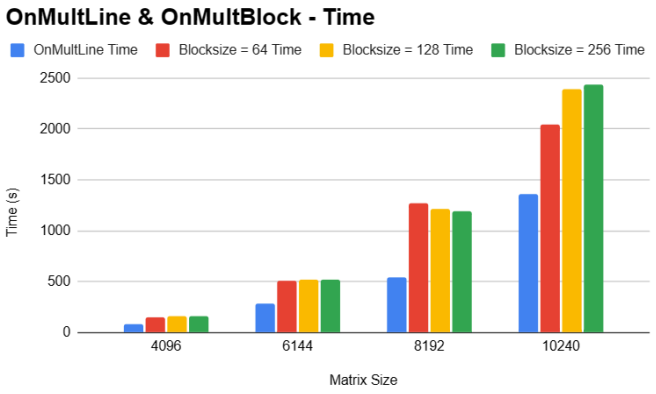

Figure 3: OnMultLine and OnMultBlock Time Comparison

When choosing a block size of 64 for the OnMultBlock algorithm, there's an increase in both L1 and L2 cache misses and execution time compared to the OnMultLine algorithm. A possible reason for this is the overhead of managing blocks in OnMultBlock.

We can conclude that the optimal block size can vary depending on factors such as the size of the matrices and the size of the cache.

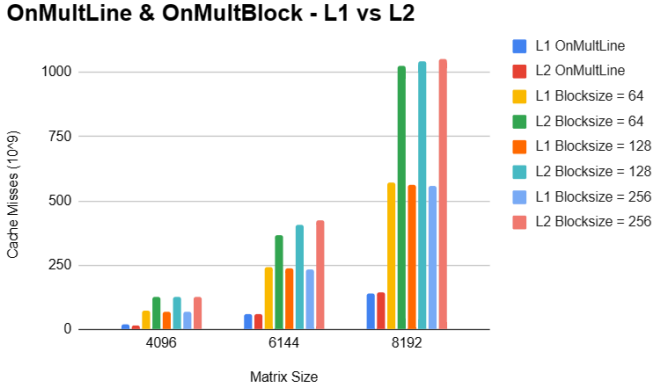

Figure 4: OnMultLine and OnMultBlock Cache Misses Comparison

Upon analyzing the OnMultBlock cache misses for different block sizes (64, 128, and 256), we observe that cache misses don't significantly increase when the block size is incremented by multiples of 64. This could be because the size of a cache line (the unit of data transfer between the cache and main memory) in modern processors is typically 64 bytes. When the block size aligns with the cache line size, it can lead to efficient cache usage. This is because block matrix multiplication is designed to leverage spatial locality, working with blocks of data that fit into the cache.

Part 2: Performance Evaluation of a Multi-Core Implementation

So far we have experimented with sequential computing where operations are performed one after the other. It does not fully utilize the computing power of modern systems. In contrast, parallel computing involves dividing a problem into smaller subproblems that can be solved simultaneously. The main advantage of parallel computing is that it can significantly speed up the execution time for large-scale computations.

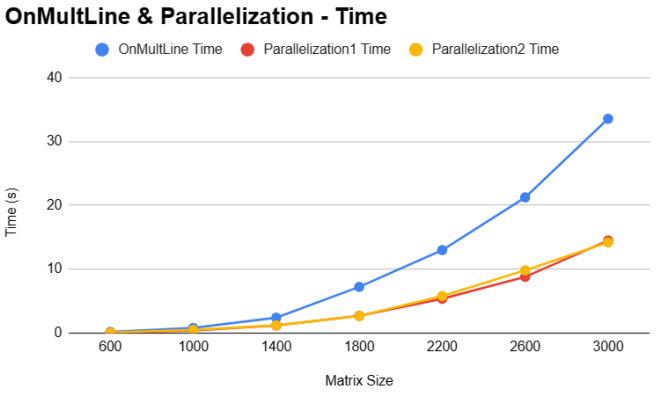

Figure 5: OnMultLine and Parallelization Time Comparison

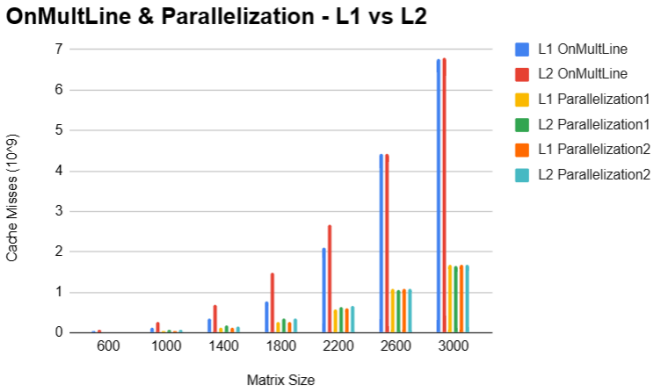

Figure 6: OnMultLine and Parallelization Cache Misses Comparison

In our observations, we found that the parallelized algorithms experienced fewer cache misses compared to the sequential ones. This improvement is likely achieved through cache partitioning and prefetching, resulting in decreased execution time.

Given each core typically possesses its own private cache, data is distributed across them. This increases the total cache size available to the program thereby reducing cache misses and ultimately leading to faster execution times.

Modern processors employ prefetching, a technique where they anticipate the data needed next and preload it into the cache in advance. When tasks are independent and their data access patterns are predictable, prefetching can lead to a decrease in cache misses.

Parallelization1 (P1) and Parallelization2 (P2) algorithms refer to using parallel computing in the outermost loop and both outermost and innermost loops, respectively.

P1 creates a higher level of parallelism because each iteration of the outer loop can potentially run on a different core. This leads to better load balancing and utilization of multiple cores, which in turn translates to a noticeable speedup for large matrices. However, the speedup is highly dependent on cache misses. If different iterations of the outer loop work on different data that can't fit into the cache simultaneously, the speedup suffers. This is especially true if the data accessed in each iteration of the outer loop is far apart in memory, leading to poor spatial locality. The GFLOPs rate is also higher due to the increased computational throughput.

P2 is an approach that can potentially reduce cache misses because each thread can reuse the data loaded into the cache by the outer loops. This takes advantage of temporal locality, where if a location is referenced, it will tend to be referenced again soon. However, it might lead to less efficient utilization of multiple cores if the number of iterations in the inner loop is small. Also, synchronization overhead can be

higher if the innermost loop has fewer iterations. P2 might show a smaller speedup for small matrices, due to the lower level of parallelism and potential synchronization overhead. However, the efficiency might be higher if the increased data reuse leads to a lower number of cache misses, which is the case when the matrix size increases. The GFLOPs rate might be lower or similar, depending on the balance between computation and memory access.

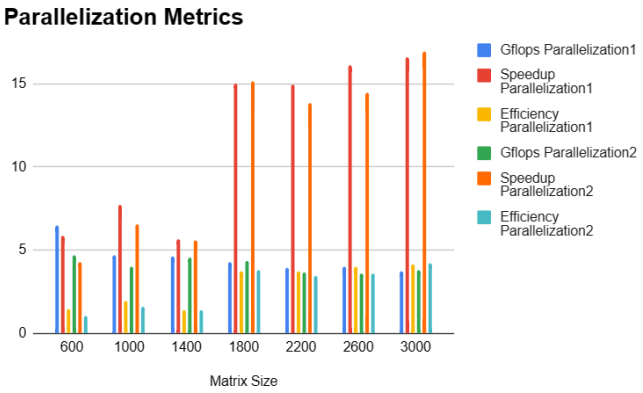

Figure 7: Parallelization Metrics

To uniformly compare these algorithms with OnMult, we utilized specific metrics such as GFLOPs, Speedup, and Efficiency. GFLOPs represent the number of floating-point operations performed per second. Speedup is the ratio of the execution time of OnMult to the execution time of the parallelized algorithm. Lastly, Efficiency is calculated by dividing the Speedup by the number of cores, which we set to 4 for our tests.

Conclusions

We conclude by stating that optimizing matrix multiplication algorithms and leveraging parallel computing techniques can lead to improved performance by reducing cache misses and utilizing computing resources more efficiently. Additionally, understanding cache behavior and aligning algorithmic choices with cache architecture can further enhance performance in computational tasks.

Appendix 1 - Implemented Algorithms

OnMult Algorithm

```
for(i=0; i<m_ar; i++){
    for( j=0; j<m_br; j++){
        temp = 0;
        for( k=0; k<m_ar; k++)
            temp += pha[i*m_ar+k] * phb[k*m_br+j];
        phc[i*m_ar+j]=temp;
    }
}
```

OnMultLine Algorithm

```
for(i=0; i<m_ar; i++){
    for(k=0; k<m_ar; k++){
        for( j=0; j<m_br; j++)
            phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
    }
}
```

OnMultBlock Algorithm

```
for (i = 0; i < m_ar; i += bkSize) {
    for (j = 0; j < m_ar; j += bkSize) {
        for (k = 0; k < m_ar; k += bkSize) {
            // Perform block matrix multiplication
            for (int ii = i; ii < i + bkSize && ii < m_ar; ii++) {
                for (int jj = j; jj < j + bkSize && jj < m_ar; jj++) {
                    for (int kk = k; kk < k + bkSize && kk < m_ar; kk++) {
                        phc[ii * m_ar + jj] += pha[ii * m_ar + kk] * phb[kk * m_br + jj];
                    }
                }
            }
        }
    }
}
```

Appendix 2 - Result Tables for Part 1

Part 1: Single Core Performance

OnMult (*)

Matrix Size	C++ Time	Java Time	L1	L2
600	0,392	0,471	244284072	38930669
1000	3,265	4,337	1305738713	269520415
1400	6,749	10,160	3538698911	1392858422
1800	40,558	45,954	9252639030	9182879228b
2200	80,297	103,720	17948440202	24594531498
2600	141,923	173,841	31877668175	59468915569
3000	240,156	290,026	50660148140	105015838297

OnMultLine (*)

Matrix Size	C++ Time	Java Time	L1	L2
600	0,195	0,297	27325414	56948161
1000	0,804	1,260	126773322	258530231
1400	2,415	3,454	352055194	705130339
1800	7,230	7,427	765003369	1476164985
2200	12,986	18,442	2103924207	2657827270
2600	21,241	34,217	4412276321	4423934826
3000	33,546	52,833	6782852342	6790975239
4096	86,076		17649221425	17383010029
6144	285,921		59573687598	59353785673
8192	537,485		141429844139	143710858133
10240	1365,890		275535323922	284332976295

OnMultBlock (*) (**)

- Blocksize = 64

Matrix Size	C++ Time	L1	L2
4096	154,848	71232656978	124705115066
6144	505,406	240175295098	365143168043
8192	1273,84	570078915600	1024571308086
10240	2047,715		

- Blocksize = 128

Matrix Size	C++ Time	L1	L2
4096	157,150	70055586009	127527984234
6144	520,084	236342795224	408277913993
8192	1213,54	560555527880	1042701015607
10240	2389,538		

- Blocksize = 256

Matrix Size	C++ Time	L1	L2
4096	158,301	69535675568	128071669525
6144	518,873	234485719071	425046177601
8192	1187,200	557464655972	1051796267405
10240	2436,854		

Notes:
(*) For smaller-sized matrices (600 to 3000), we ran each test 20 times. For bigger-sized matrices (4096 to 10240), we repeated it 5 times for time efficiency reasons.
(**) For a matrix size of 10240, we ran the OnMultBlock algorithm just once, opting against utilizing Papi due to excessive time consumption.

Appendix 3 - Result Tables for Part 2

Part 2: Multi-Core Performance

Parallelization on the outermost loop (P1)

Matrix Size	Time	L1	L2	Gflops	Speedup	Efficiency (cores = 4)
600	0,067	6843225	14199889	6,458	5,867	1,467
1000	0,424	31877875	62687437	4,718	7,701	1,925
1400	1,190	113341556	168990063	4,610	5,669	1,417
1800	2,702	260884598	354557405	4,316	15,008	3,752
2200	5,375	591897354	647905527	3,962	14,939	3,735
2600	8,812	1089550339	1064872172	3,989	16,106	4,026
3000	14,487	1685572898	1666902905	3,727	16,576	4,144

Parallelization on both loops (P2)

Matrix Size	Time	L1	L2	Gflops	Speedup	Efficiency (cores = 4)
600	0,091	6786057	13482765	4,731	4,298	1,075
1000	0,499	32155380	62502224	4,008	6,543	1,636
1400	1,207	111892718	167006177	4,547	5,592	1,398
1800	2,674	255725719	356728858	4,362	15,168	3,792
2200	5,793	595552437	663142728	3,676	13,861	3,465
2600	9,818	1093560172	1099543397	3,580	14,455	3,614
3000	14,205	1684164190	1675652156	3,801	16,906	4,227