Performance Evaluation for Single Threaded Matrix Multiplication Algorithms

João Pereira up202007145@fe.up.pt
Mariana Rocha up202004656@fe.up.pt
Matias Vaz up201900194@fc.up.pt
DEI, FEUP
Porto, Portugal

ABSTRACT

This report was made in the context of the Parallel and Distributed Computing (CPD) course unit to solidify the course content with practice. The main goal of the project was to develop, compare, and document the code for matrix multiplication in single-threaded applications.

The idea behind the practical work was to learn about the CPU performance impact of accessing large amounts of data in memory. Thus, we compared different algorithms with different ways to optimize memory access in C++ and Python.

1 PROBLEM DESCRIPTION

The main problem for the project was evaluating the performance of different algorithms. The main difference for the given algorithms isn't the result itself, nor a math insight for general use, but a computation optimization for accessing the memory fewer times or in a faster manner.

For this study, three algorithms were used to evaluate the costs that different approaches would have on performance in a single-threaded environment. We recreated these algorithms in C++ and Python and compared different sizes for square matrices.

We chose these languages based on their current usage. Since C++ is a compiled fast-performance programming language also used for machine learning algorithms, which extensively use matrix multiplications. The other one is Python, a popular language for beginners in programming, with a more friendly and soft learning curve. It is also used extensively for data science problems, which use matrix multiplications.

2 ALGORITHMS EXPLANATION

2.1 Naive Matrix Multiplication

The naive matrix multiplication algorithm was provided in a C++ program and later our group implemented the same algorithm in the Python language.

The algorithm operates as follows: the result in the i-th row and j-th column of matrix **C** is the outcome of the dot product between the elements in the i-th row of matrix **A** and the j-th column of matrix **B**. This results in a time complexity of $O(2n^3)$.

```
for i in range(0, matrix_size, 1):
  for j in range(0, matrix_size, 1):
    for k in range(0, matrix_size, 1):
      matrix_c[i][j] += matrix_a[i][k] *
          ↪ matrix_b[k][j]
```

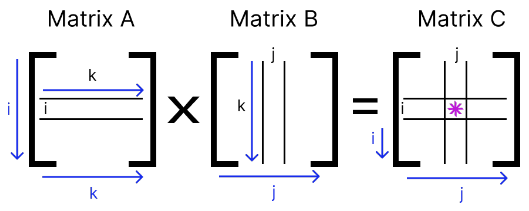


Figure 1: Naive Matrix Multiplication algorithm illustration

2.2 Lines Matrix Multiplication

This algorithm differs from the one presented above in the order of the second and third `for` loops.

The algorithm operates as follows: the result in the i-th row and j-th column of matrix **C** is also the outcome of the dot product between the elements in the i-th row of matrix **A** and the j-th column of matrix **B**. It differs from the former because the whole dot product is not calculated at once. It multiplies an element of **A** with a row of **B**, accumulating the result in the correct row of **C**. This results in a time complexity of $O(2n^3)$.

```
for i in range(0, matrix_size, 1):
  for k in range(0, matrix_size, 1):
    for j in range(0, matrix_size, 1):
      matrix_c[i][j] += matrix_a[i][k] *
          ↪ matrix_b[k][j]
```

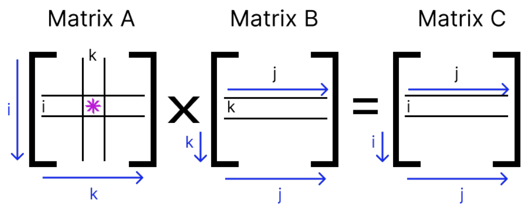


Figure 2: Lines Matrix Multiplication algorithm illustration

2.3 Block Matrix Multiplication

The block matrix algorithm works as follows: firstly, both matrices to be multiplied are divided into blocks of size s, resulting in n blocks. Those blocks are treated as elements of a matrix and the lines matrix multiplication algorithm is used to multiply them. This results in a time complexity of $O(2n^3)$.

João Pereira up202007145@fe.up.pt
Mariana Rocha up202004656@fe.up.pt
Matias Vaz up201900194@fc.up.pt

```
for ib in range(0, n, 1):
  for kb in range(0, n, 1):
    for jb in range(0, n, 1):
      for i in range(ib * s, (ib + 1) * s, 1):
        for k in range(kb * s, (kb + 1) * s, 1):
          for j in range(jb * s, (jb + 1) * s, 1):
            matrix_c[i][j] += matrix_a[i][k] *
                ↪ matrix_b[k][j]
```

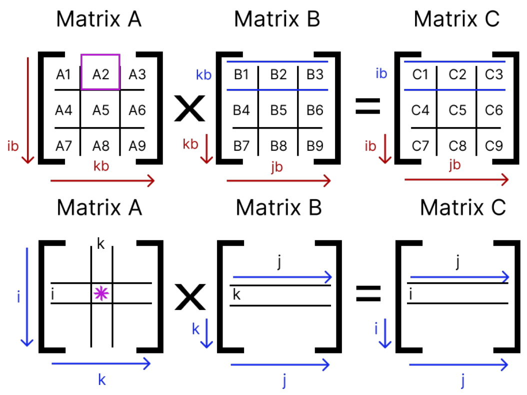


Figure 3: Block Matrix Multiplication algorithm illustration

3 EXPERIMENTAL SETUP

Since every different hardware has its own peculiarities, it's important to share the details of the setup used for this project. It's important to mention, even with being able to easily parallelize the code with some of our tools, we decided to focus only on single-thread applications. After that, we will discuss the experimental results found.

3.1 Computer Environment Setup

For every scenario, a desktop computer running Ubuntu 22.04 with 32 GB of ram and an AMD Ryzen 5 5600x was used to process everything. Besides the terminal, no other essential process was open, minimizing the noise of other programs in our data.

For collecting the data cache misses, we used the package PAPI 7.0.0, a Linux tool developed by the University of Tennessee.

3.2 Code Environment Setup

For programming, we used C++ 11 with GCC 11.3.0, Python 3.9.13, PyPy 7.3.9, and Numba 0.55.1. We chose these tools because of their broad use. Pypy is a Just In Time compiler for Python, and Numba is a framework that compiles and allows easy parallelization of Python code, using GPUs or CPUs. Both tools hugely speed up the runtime.

The decision for using all these tools was supported by the trade-off between effort in development versus performance. Using more abstractions often results in an easier process to develop, but hurts the resulting performance.

4 EXPERIMENTAL RESULTS

The experimental results were surprising. Regardless of the gain in performance related to better memory access, we got some interesting results we'd like to point out.

First of all, the block algorithms didn't show any kind of improvement in performance in our tests. That wasn't our first assumption. We could also notice all block sizes having relatively close runtimes.

In 4.1 and 4.2, we will compare the time performance between Python and C++, and will analyse the memory performance for C++. In 4.3, we will analyse the performance in C++ only. And in 4.4, we will give a brief comparison between all the algorithms and languages. You can find tables with all the results in the Appendix.

4.1 Basic Multiplication Performance

In figure 4, we can see how slow Python is compared to C++. In a square matrix with size 3072, we got a runtime of 4692.512 s, versus 149.787 s for C++. That means C++ had 31.32 times better performance.

You can also see the difference between C++ and the alternative Python implementations. PyPy and C++ had similar performance in our tests, being capable of running with a size of 3072 in around 150 s. The Numba code, without any kind of parallelization, is a little bit slower compared to the other two tools.

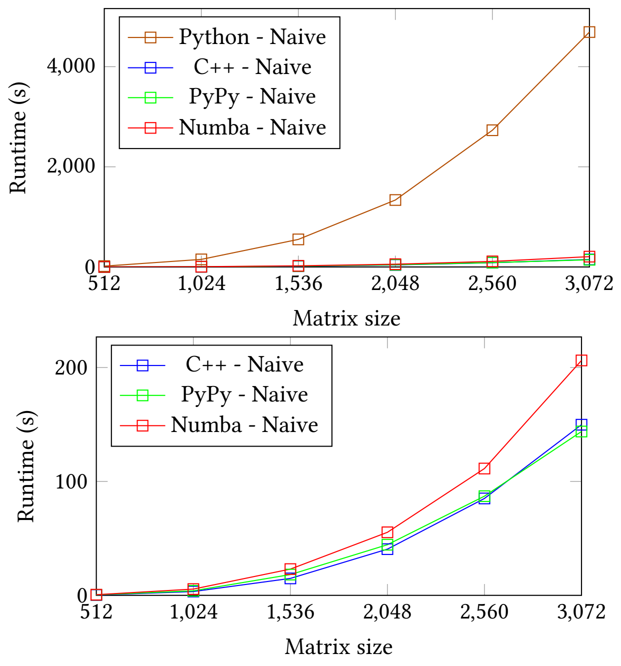


Figure 4: Runtime for the Basic algorithm

4.2 Lines Multiplication Performance

In figure 5, we once again see how slow Python is compared to C++. In a square matrix with size 3072, we got a runtime of 3679.208 s, versus 10.874 s for C++. That means C++ had 338.35 times better performance.

You can also see the contrast between C++ and the alternative Python implementations. PyPy had almost exactly the same performance as in 4.1, while C++ and Numba had better performance, with runtimes of around 11 s and 40 s, respectively.

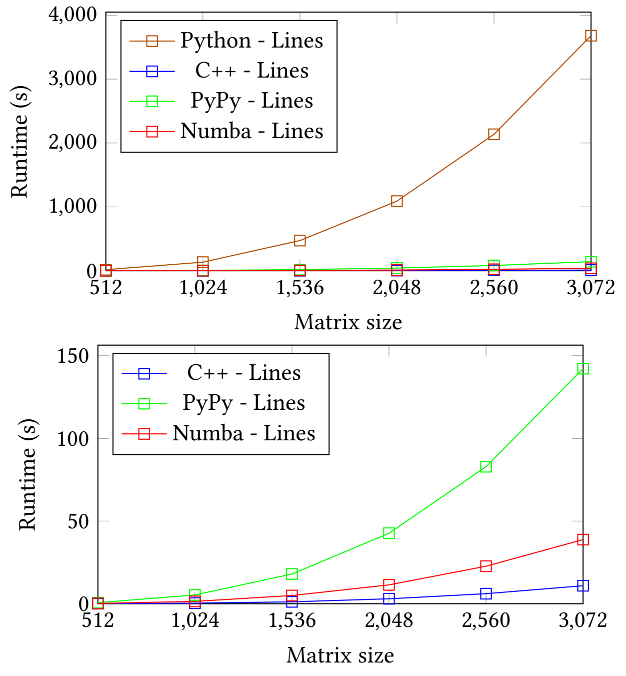


Figure 5: Runtime for the Lines algorithm

4.3 Block Multiplication Performance

In figure 6, it is possible to see the runtime for the Blocks algorith with different block sizes. One interesting thing to notice is that the block size used had little to no impact on the overall performance. This observation would probably change in parallelized code, but any predictions as to which size would be best would be pure conjecture, as there are many variables that could affect this outcome.

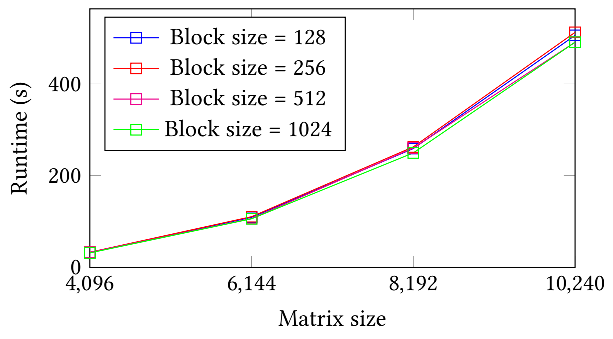


Figure 6: Runtime for the Blocks algorithm

4.4 Overall Comparison

All tested algorithms were slower when using a Python implementation compared to C++. This is due to the fact that Python is an interpreted language, which greatly increases the runtime for any algorithm. When using PyPy, a Just In Time compiler, this performance hit largely goes away, but it is still slower than when using C++, as the compilation happens at runtime, and not earlier. Interestingly, PyPy gains almost no performance when switching to the Lines algorithm, this could suggest that it optimizes the Naive algorithm by itself. The Numba library for Python brought its performance much closer to C++ than PyPy, but still about four times slower.

Comparing the Naive algorithm with the Lines algorithm, all implementations get a speedup in performance. This can be attributed to a better memory access strategy that minimizes cache misses, which are very expensive. As seen in figures 7 and 8, the Lines algorithm reduces L1 cache misses by an order of magnitude and L2 cache misses by up to three orders of magnitude in larger matrices.

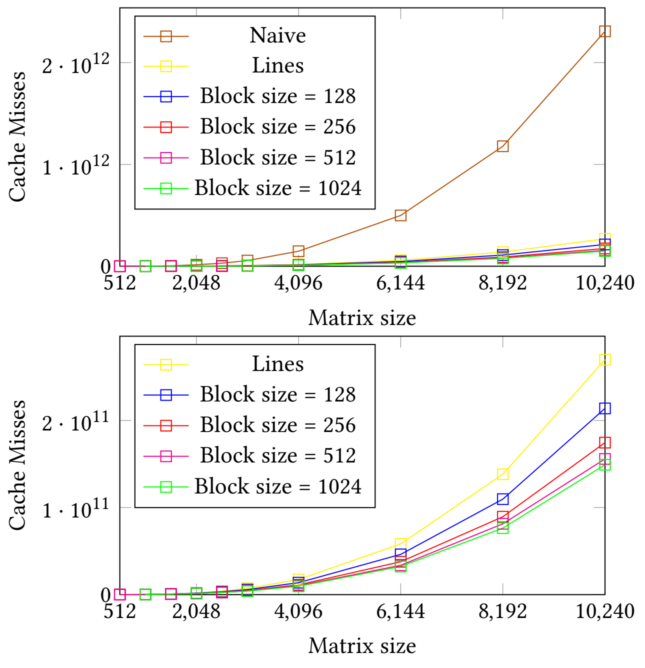
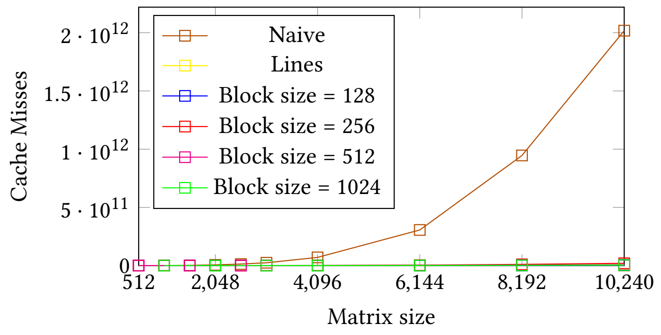


Figure 7: L1 data cache misses for C++ Algorithms

Cache misses are minimized because, as seen in figure 2, the inner loop does not change which row of the matrices we're accessing. This is beneficial because our matrices are stored as a contiguous array in memory, with the rows next to each other, and CPUs pull contiguous blocks of memory into the cache at a time. So "jumping around" in memory, as we do when changing rows, is very expensive in terms of memory access.

In figure 9, we can see the resulting performance for each of the algorithms. It stays mostly constant for each algorithm, except for small matrices. Here the Naive algorithm is slightly faster, as the amount of cache misses is also smaller when compared to the computations themselves. The Lines algorithm gets a bump around a matrix size of 1024, this is the point where our program reaches the best balance between smaller matrices, where overhead has a bigger impact, and larger matrices, where loading data from memory has a bigger impact. The Blocks algorithm is slightly slower at smaller



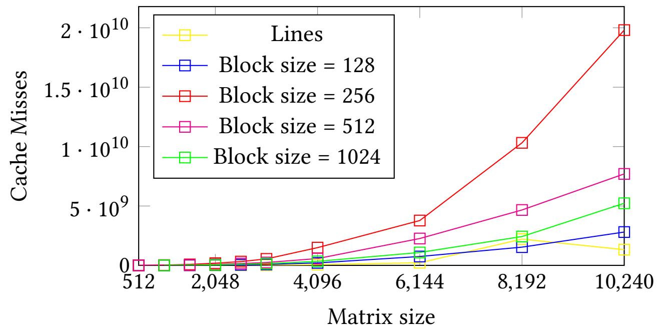


Figure 8: Cache misses on L2 for C++ Algorithms

sizes, probably due to the overhead that comes with splitting those matrices unnecessarily.

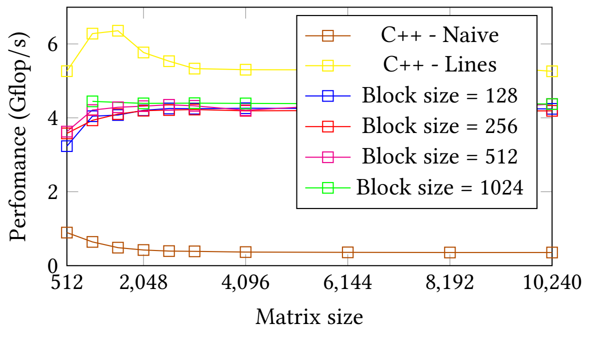


Figure 9: Performance of different algorithms

This chart reinforces what was said in previous sections, that the Lines algorithm is faster than the Naive algorith, and the Blocks algorithm is slightly slower than the Lines algorithm, at least in a single threaded application.

Additional data not pictured in these figures can be found in the Appendix.

5 RELATED WORK

In future work, we could parallelize the code using CPUs and GPUs and compare the different approaches. The usage of multi-processing methods would be the main way to gain performance, which would provide a better understanding of the memory impact and performance possibilities.

Besides this, exploring more algorithms to eventually find an even more efficient one in terms of memory access could also be an option.

6 CONCLUSIONS

This practical assignment was instrumental in a better understanding of the importance of efficiently accessing memory in programs according to their hardware design and its impact on overall performance.

It was also important for realizing just how much slower Python is compared to C++, and that there are faster alternatives that improve its performance without sacrificing its user-friendly characteristics, namely PyPy and Numba.

To conclude, choosing the most performant algorithm goes further than complexity analysis. Thinking about the programming language, the hardware and the specific application of the algorithm is also important. In some areas, such as machine learning or scientific research, differences like these that may seem small can have a big difference, as they work with large amounts of data and low latency is a critical requirement.

ACKNOWLEDGMENTS

We want to thank all the staff of the Parallel and Distributed Computing course unit, especially our practical lesson teacher, Carlos Baquero-Moreno.

A APPENDIX

Matrix size	C++						Python		PyPy		Numba	
	Naive	Lines	Blocks				Naive	Lines	Naive	Lines	Naive	Lines
			128	256	512	1024						
512	0.300	0.051	0.083	0.075	0.074	–	17.070	16.820	0.475	0.661	0.580	0.181
1,024	3.343	0.342	0.531	0.546	0.510	0.483	151.403	134.520	3.973	5.289	5.515	1.403
1,536	14.878	1.140	1.778	1.767	1.691	–	550.667	472.742	18.231	17.966	23.052	4.947
2,048	40.557	2.978	4.087	4.097	3.980	3.912	1,337.473	1,089.743	44.461	42.625	55.349	11.393
2,560	85.065	6.064	7.888	7.968	7.709	–	2,731.605	2,135.442	87.082	82.878	111.327	22.684
3,072	149.787	10.874	13.653	13.725	13.405	13.184	4,692.512	3,679.208	143.811	142.200	206.206	38.779
4,096	373.929	25.917	32.258	32.799	32.683	31.304	–	–	–	–	–	–
6,144	1,288.833	87.554	109.109	110.443	106.283	105.837	–	–	–	–	–	–
8,192	3,091.237	201.778	259.081	262.723	260.295	249.617	–	–	–	–	–	–
10,240	6,035.115	408.235	506.042	512.623	491.129	491.097	–	–	–	–	–	–

Table 1: Runtime for all algorithms (s)

Matrix size	C++						Python		PyPy		Numba	
	Naive	Lines	Blocks				Naive	Lines	Naive	Lines	Naive	Lines
			128	256	512	1024						
512	0.895	5.263	3.234	3.579	3.628	–	0.016	0.016	0.565	0.406	0.463	1.483
1,024	0.642	6.279	4.044	3.933	4.211	4.446	0.014	0.016	0.541	0.406	0.389	1.531
1,536	0.487	6.358	4.076	4.102	4.286	–	0.013	0.015	0.398	0.403	0.314	1.465
2,048	0.424	5.769	4.204	4.193	4.317	4.392	0.013	0.016	0.386	0.403	0.310	1.508
2,560	0.394	5.534	4.254	4.211	4.353	–	0.012	0.016	0.385	0.405	0.301	1.479
3,072	0.387	5.332	4.247	4.225	4.325	4.398	0.012	0.016	0.403	0.408	0.281	1.495
4,096	0.368	5.303	4.261	4.190	4.205	4.390	–	–	–	–	–	–
6,144	0.360	5.298	4.251	4.200	4.364	4.383	–	–	–	–	–	–
8,192	0.356	5.449	4.244	4.185	4.224	4.405	–	–	–	–	–	–
10,240	0.356	5.260	4.244	4.189	4.373	4.373	–	–	–	–	–	–

Table 2: Efficiency all algorithms (Gflop/s)

Matrix size	Naive	Lines	Blocks			
			128	256	512	1024
512	$1.447 \cdot 10^{8}$	$1.745 \cdot 10^{7}$	$2.673 \cdot 10^{7}$	$2.196 \cdot 10^{7}$	$1.736 \cdot 10^{7}$	–
1,024	$1.155 \cdot 10^{9}$	$1.513 \cdot 10^{8}$	$2.149 \cdot 10^{8}$	$1.758 \cdot 10^{8}$	$1.572 \cdot 10^{8}$	$1.408 \cdot 10^{8}$
1,536	$3.914 \cdot 10^{9}$	$5.729 \cdot 10^{8}$	$7.243 \cdot 10^{8}$	$5.908 \cdot 10^{8}$	$5.270 \cdot 10^{8}$	–
2,048	$1.436 \cdot 10^{10}$	$1.510 \cdot 10^{9}$	$1.712 \cdot 10^{9}$	$1.396 \cdot 10^{9}$	$1.249 \cdot 10^{9}$	$1.185 \cdot 10^{9}$
2,560	$3.020 \cdot 10^{10}$	$4.045 \cdot 10^{9}$	$3.333 \cdot 10^{9}$	$2.725 \cdot 10^{9}$	$2.432 \cdot 10^{9}$	–
3,072	$5.549 \cdot 10^{10}$	$7.215 \cdot 10^{9}$	$5.771 \cdot 10^{9}$	$4.719 \cdot 10^{9}$	$4.213 \cdot 10^{9}$	$4.033 \cdot 10^{9}$
4,096	$1.476 \cdot 10^{11}$	$1.730 \cdot 10^{10}$	$1.371 \cdot 10^{10}$	$1.121 \cdot 10^{10}$	$1.018 \cdot 10^{10}$	$9.568 \cdot 10^{9}$
6,144	$4.989 \cdot 10^{11}$	$5.831 \cdot 10^{10}$	$4.624 \cdot 10^{10}$	$3.774 \cdot 10^{10}$	$3.369 \cdot 10^{10}$	$3.230 \cdot 10^{10}$
8,192	$1.178 \cdot 10^{12}$	$1.383 \cdot 10^{11}$	$1.097 \cdot 10^{11}$	$8.947 \cdot 10^{10}$	$8.131 \cdot 10^{10}$	$7.641 \cdot 10^{10}$
10,240	$2.307 \cdot 10^{12}$	$2.698 \cdot 10^{11}$	$2.139 \cdot 10^{11}$	$1.746 \cdot 10^{11}$	$1.558 \cdot 10^{11}$	$1.492 \cdot 10^{11}$

Table 3: L1 data cache misses for C++ algorithms

João Pereira up202007145@fe.up.pt
Mariana Rocha up202004656@fe.up.pt
Matias Vaz up201900194@fc.up.pt

Matrix size	Naive	Lines	Blocks			
			128	256	512	1024
512	$3.405 \cdot 10^6$	$1.043 \cdot 10^5$	$2.081 \cdot 10^5$	$2.477 \cdot 10^6$	$1.062 \cdot 10^5$	–
1,024	$2.805 \cdot 10^8$	$4.942 \cdot 10^5$	$2.084 \cdot 10^6$	$2.424 \cdot 10^7$	$1.016 \cdot 10^7$	$1.165 \cdot 10^6$
1,536	$2.318 \cdot 10^9$	$1.228 \cdot 10^6$	$9.932 \cdot 10^6$	$8.191 \cdot 10^7$	$3.249 \cdot 10^7$	–
2,048	$6.062 \cdot 10^9$	$2.646 \cdot 10^6$	$2.323 \cdot 10^7$	$1.840 \cdot 10^8$	$8.401 \cdot 10^7$	$4.538 \cdot 10^7$
2,560	$1.334 \cdot 10^{10}$	$8.035 \cdot 10^6$	$4.246 \cdot 10^7$	$3.181 \cdot 10^8$	$1.531 \cdot 10^8$	–
3,072	$2.440 \cdot 10^{10}$	$2.217 \cdot 10^7$	$7.800 \cdot 10^7$	$5.450 \cdot 10^8$	$2.432 \cdot 10^8$	$1.471 \cdot 10^8$
4,096	$7.048 \cdot 10^{10}$	$6.271 \cdot 10^7$	$2.090 \cdot 10^8$	$1.494 \cdot 10^9$	$5.738 \cdot 10^8$	$3.374 \cdot 10^8$
6,144	$3.060 \cdot 10^{11}$	$2.207 \cdot 10^8$	$7.480 \cdot 10^8$	$3.775 \cdot 10^9$	$2.261 \cdot 10^9$	$1.086 \cdot 10^9$
8,192	$9.461 \cdot 10^{11}$	$2.211 \cdot 10^9$	$1.536 \cdot 10^9$	$1.032 \cdot 10^{10}$	$4.664 \cdot 10^9$	$2.428 \cdot 10^9$
10,240	$2.018 \cdot 10^{12}$	$1.322 \cdot 10^9$	$2.806 \cdot 10^9$	$1.981 \cdot 10^{10}$	$7.701 \cdot 10^9$	$5.227 \cdot 10^9$

Table 4: L2 data cache misses for C++ algorithms