University of Porto - Faculty of Engineering

Master in Informatics and Computing Engineering

Project Report

Evalutation of Energy Consumption in Matrix Multiplication

Miguel Rodrigues
up201906042@edu.fe.up.pt
Sérgio Estêvão
up201905680@edu.fe.up.pt

February 2023

Contents

List of Figures

1 Introduction

According to the authors of [1], writing sustainable, power-efficient software requires a deep understanding of the power consumption behavior of a computer program. The benefit is that developers, by improving source code's efficiency, can optimize the power consumption of software. Energy efficiency can be improved by taking into account the cache behavior in read/write operations and the energy expenditure of such operations during a program's execution.

This report will cover three different approaches to matrix multiplications used to analyze the energy consumption at the CPU level. All the results and findings will be detailed ahead.

2 Algorithms explanation

The matrix multiplication algorithm (in its naïve form) is a cubic algorithm - $O(n^3)$, regarding both spatial and temporal complexity.

The essence of the algorithm is to have three different iterators, two of them to scan the rows and columns, and the third one to perform the dot product between two *vectors*.

For this report, we will focus on the naïve algorithm with distinct approaches regarding cache friendliness and parallelism. We have opted to use C++20 for implementing the approaches listed in the the following subsections due to its compatibility with tools such OpenMP and to allow a more granular refinement over the program's performance.

2.1 Column Oriented Multiplication

Column-oriented matrix multiplication: multiplies one line of the first matrix by each column of the second matrix.

```
1  using matrix_t = double[];
2  using matrix_size_t = std::size_t;
3  static constexpr matrix_size_t N = 2000;
4
5  matrix_t mult(const matrix_t a, const matrix_t b)
6  {
7      matrix_t c = new double[N * N];
8
9      for (matrix_size_t i = 0; i < N; i++)
10         for (matrix_size_t j = 0; j < N; j++)
11             for (matrix_size_t k = 0; k < N; k++)
12                 c[i * N + j] += a[i * N + k] * b[k * N + j];
13     return c;
14 }
```

2.2 Row Oriented Multiplication

Row-oriented matrix multiplication: multiplies an element from the first matrix by the correspondent line of the second matrix.

It achieves the same result as in *2.1*. However, the methodology is significantly different since the final value of an element results in an accumulation during the algorithm's execution rather than

being computed only once. This is, ultimately a cache friendly approach that takes advantage of locality, thus reducing the run-time significantly.

```
using matrix_t = double[];
using matrix_size_t = std::size_t;
static constexpr matrix_size_t N = 2000;

matrix_t mult(const matrix_t a, const matrix_t b)
{
    matrix_t c = new double[N * N];

    for (matrix_size_t i = 0; i < N; i++)
        for (matrix_size_t k = 0; k < N; k++)
            for (matrix_size_t j = 0; j < N; j++)
                c[i * N + j] += a[i * N + k] * b[k * N + j];
    return c;
}
```

2.3 Parallel Row Oriented Multiplication

As it is visible this approach takes advantage *OpenMP* to parallelize the computation effort. The approach is the same taken as in *2.2*.

```
using matrix_t = double[];
using matrix_size_t = std::size_t;
static constexpr matrix_size_t N = 2000;

matrix_t mult(const matrix_t a, const matrix_t b)
{
    matrix_t c = new double[N * N];
    matrix_size_t i, j, k;

    #pragma omp parallel for private(i, j, k) shared(a, b, c)
    for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
            for (j = 0; j < N; j++)
                c[i * N + j] += a[i * N + k] * b[k * N + j];
    return c;
}
```

3 Performance Measurements and Metrics

In order to evaluate the performance of each algorithm we used the time elapsed, energy consumption, and the machine's capacity.

$$flops = 2 * N^3 \tag{1}$$

$$capacity = \frac{flops}{time_{execution}} \tag{2}$$

The equation (1) demonstrates the number of floating-point operations performed by the algorithm, being N the size of the matrices being multiplied. Given that, we are able to compute the machine's capacity according to the equation (2).

3.1 PAPI

PAPI[1] [2], short for *Performance Application Programming Interface*, is a very useful and easy-to-use library to measure the performance of a given piece of code. Thanks to it, we were able to measure the number of cache misses - for both L1 and L2 - as well as the total number of clock cycles and instructions performed during execution.

3.2 `perf`

The `perf`[2] command [3] was used to assess the energy consumption while running the algorithm. This utility serves as a key gateway to harness the performance monitoring capabilities of the *Linux* kernel.

Moreover, Intel processors can obtain an estimation for energy consumption through an API called RAPL [4], short for *Running Average Power Limit*.

Considering this, `perf` uses RAPL which exposes the processor's counters to read and make estimations about energy consumption. It is important to notice that, in spite of being estimations, this is at the moment the most reliable way to track the energy expenditure of a particular execution of a program.

Additionally, the usage of `perf`, in the context of this project, was pretty straightforward. The command `perf stat -e power/energy-cores/ <executable>` executes a given program and when finished it gives a very detailed report of the energy expenditure in Joules.

4 Results

These results were obtained by calculating the average between of 3 distinct executions for each algorithm and each matrix size, in a machine with a 5.15.0-60-generic Kernel, running an Intel® Core™ i7-9750H CPU (Coffee Lake) with 6 cores and 12 threads with Hyper-Threading @ 2.60 GHz.

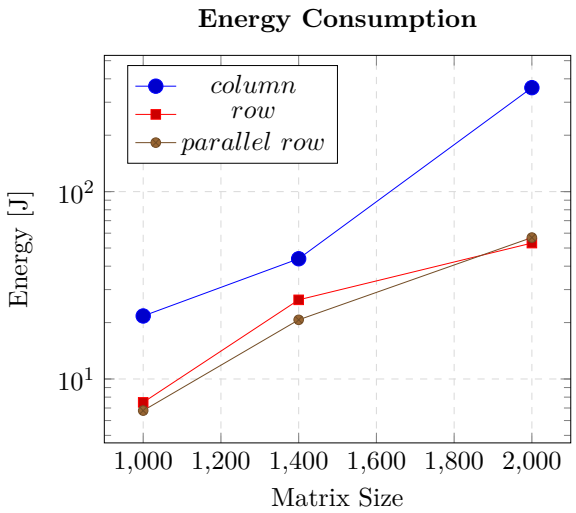
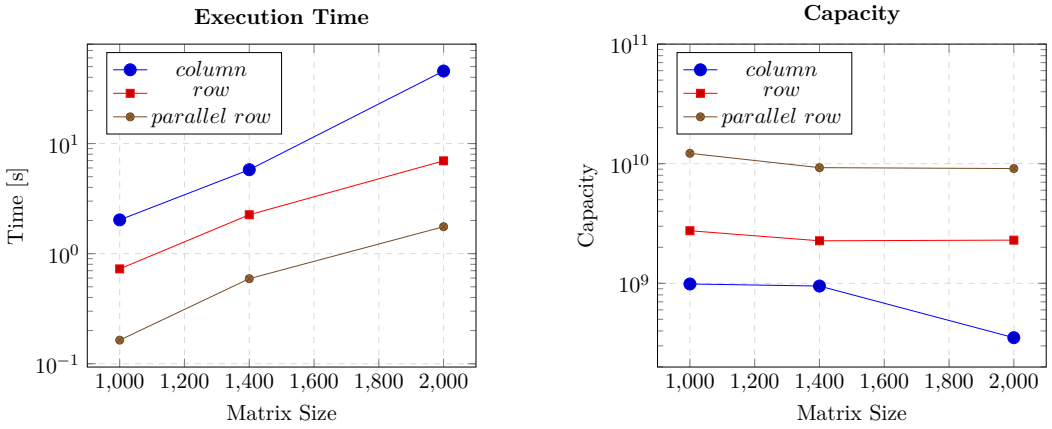


Figure 1: Processor's energy expenditure while executing matrix multiplication

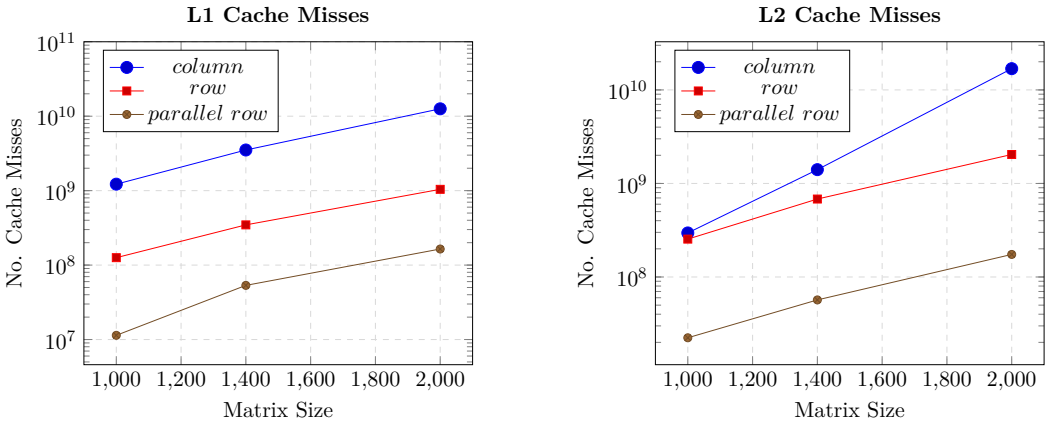
[1] The version 7.0.0.0 of PAPI was used.
[2] `perf`'s version is associated with the version of the *Linux* kernel being used.



(a) Execution time for matrix multiplication

(b) Machine's capacity

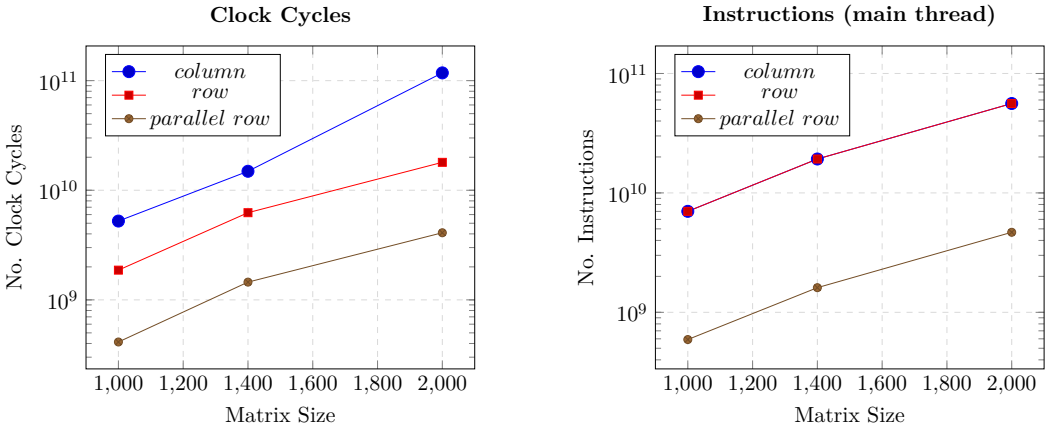
Figure 2: Traditional metrics for matrix multiplication performance assessment



(a) Cache level 1 data misses

(b) Cache level 2 data misses

Figure 3: Cache data misses during matrix multiplication execution



(a) Total of clock cycles completed

(b) Total number of instructions executed

Figure 4: Absolute metrics for matrix multiplication performance assessment

4.1 Analysis

After analyzing the results, we verified, as expected, that there is a direct relation between cache conscious approach and energy consumption.

The energy efficiency of row oriented approaches - both for the sequential and parallel environments - outperformed the column oriented matrix multiplication.

In *figure 1*, this can be justified by the number of L1 and L2 cache misses during the execution, which implies that the program needed to read from memory and write to the cache much more frequently. Obviously, these operations induce an energy expenditure that is considerably higher when compared to only reading from the cache. Moreover, it is critical to notice, that the exchange of data between memory and the CPU's cache is very costly in terms of latency. This means that the accumulation of latency will translate into longer execution time, which in turn, translates into an increase in clock cycles - compare the similar lines of *figure 2.a.* and *figure 4.a.*.

The difference in L1 and L2 cache misses between these two different approaches exists because when reading a value from the matrix, the CPU fetches it from the cache. If the operation is unsuccessful, i.e. a data cache miss occurred, it will fetch the value from the main memory and copy a chunk of data next to that value to the cache. The new readings next to the previous values will consequently be faster since the data will more often be in the cache and therefore closer to the CPU.

With that in mind, the row algorithm is cache friendly and reads data in row oriented order, it takes advantage of the data locality meaning that the ratio of cache hits will increase, therefore the algorithm will execute in a shorter period of time and with a shorter energetic footprint.

Finally, as seen in *figure 1*, the parallel row multiplication was able to outperform the single-threaded row multiplication by an order of magnitude in terms of execution time. This was achieved while maintaining a similar energy consumption. These results are possible due to the fact that the matrix multiplication algorithm is highly parallelizable. According to Amdahl's law, the speedup obtained will be significant, since it involves multiple independent computations that can be performed simultaneously between threads and with little to no sequential tasks.

Furthermore, multi-threaded executions reduce the idle time of single-threaded ones, which means that although more threads and cores provide computational power, the speed-up balances out these extra expenditures by processing the instructions in a shorter period of time.

5 Conclusions

Through this project, we were able to study and fully comprehend the impact of data cache access on the energy consumption of algorithms and how a deficient cache access can hinder the energy efficiency of a system.

We registered these implications in real time with different approaches to the multiplication of matrices. In essence, we concluded that for highly parallelizable algorithms, a multi-threaded approach reduces the execution time while maintaining the energy consumption of a single-threaded program.

Knowledge about the cache system and its impact on the energy consumption of algorithms will undoubtedly prove very useful in developing more energy and cost efficient systems.

References

[1] H. Acar, G. I. Alptekin, J.-P. Gelas, and P. Ghodous, "The impact of source code in software on power consumption," *International Journal of Electronic Business Management*, vol. 14, pp. 42–52, 2016.

[2] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009* (M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, eds.), (Berlin, Heidelberg), pp. 157–173, Springer Berlin Heidelberg, 2010.

[3] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, pp. 1–42, 2010.

[4] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.