Project 1 - Performance evaluation of a single core

Report

Integrated Masters in Informatics and Computing Engineering

Parallel Computing

Group:

Gonçalo Moreno	up201503871	up201503871@fe.up.pt
João Mendes	up201505439	up201505439@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

April 23, 2019

Index

1 Introduction

In this project we were asked to study the effect on the processor performance of the memory hierarchy when accessing large amounts of data.

Instead of starting with making a parallel version of the algorithm, or even thinking of improving the hardware for better results, we were challenged to change the algorithm in order to take advantage of the algorithm characteristics. Therefore, we are evaluating if parallel computing is always necessary and if we can do enhancements before recurring to it.

For this study, the product of two matrices was used because it is a resource heavy operation. This requires large memory access and allows to evaluate the performance of the CPU. Memory access is quite expensive, so the CPU cache memory keeps data available faster. This advantage comes at a price, cache memory is much smaller and as to be used in an intelligent way.

The 1st algorithm is naive, doing the operation sequentially and the 2nd takes advantage of cache memory, multiplying line by line.

Cache miss occurs within cache memory access modes and methods. For each new request, the processor searched the primary cache to find that data. If the data is not found, it is considered a cache miss.

Each cache miss slows down the overall process because after a cache miss, the central processing unit (CPU) will look for a higher level cache, such as L1, L2, L3 and random access memory (RAM) for that data.

We will analyze the cache behavior through Valgrind - a programming tool for memory debugging, memory leak detection, and profiling.

2 Problem Description

In this paper we will analyze three different algorithms, each will solve the classical problem of multiplying two matrices. A recurrent problem in any scientific computing context.

$$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj},$$

for $i = 1, ..., n$ and $j = 1, ..., p$.

From here we conclude that $\mathbf{C}[i,j]$ is the sum of each element of the line i of matrix $\mathbf{A}$ multiplied by the respective element of the column j of matrix $\mathbf{B}$. Thus, the number of columns of $\mathbf{A}$ must be equal to the number of rows in $\mathbf{B}$.

In our experiments we use square matrices (NxN) which is the resulting size of matrix $\mathbf{C}$.

3 Algorithms Explanation

3.1 Algorithm 1

The first algorithm is the simplest and textbook way of multiplying matrices. This leads to an algorithm that loops over the indices i from 1 through n and j from 1 through n, computing the above formula using a nested loop:

```
def ijk(A, B, C):
    for i in range(len(A)):
        for j in range(len(A)):
            for k in range(len(A)):
                        C[i][j] += A[i][k] * B[k][j]
```

Since we are using a square matrix, we have the running time complexity $\mathcal{O}(n^3)$ and averaging $2(n^3)$ floating point operations (FLOPS).

While it is the simplest it also gives the worst performance. Due to its nature every time it accesses the **B** matrix it populates its cache with $\mathbf{B}[k][j{+}{+}]$ values which are consequently wasted. This generates a lot of cache misses and leads to poor performance.

3.2 Algorithm 2

A improved version of the first algorithm can be obtained simply by switching two of the loops. The inner most loop and the middle on.

```
def ikj(A, B, C):
    for i in range(len(A)):
        for k in range(len(A)):
            for j in range(len(A)):
                C[i][j] += A[i][k] * B[k][j]
```

This time the accesses to the second matrix **B** are more cache friendly as values in the cache can be reused leading to less cache misses and ultimately faster execution.

A problem will arise as matrix sizes increases and each line is bigger than the cache, leading to more cache misses.

3.3 Algorithm 3

This last algorithm achieves the best performance by doing the multiplications of the matrices by blocks, meaning that it can keep its CPU cache with relevant values even with massive matrix sizes.

```
for (i = 0; i < size; i += incr) {
    for (j = 0; j < size; j += incr) {
        C[i*size+j] = 0.0;
        for (k = 0; k < size; k += incr){
                for (z = k; z < std::min( k + incr, size ); z++) {
                for (x = i; x < std::min( i + incr, size ); x++) {
                        for (y = j; y < std::min( j + incr, size ); y++) {

                        C[x * size + y] += A[x * size + z] * B[z * size + y];
```

A problem that comes with this implementation is the choice of the size of blocks M. We also empirically tested the best value but theoretically it must be a small enough to fit each line of the matrix block in cache.

4 Performance Metrics

Evaluating each algorithm was done with a array of increasing matrices to understand its time complexity and how important is cache.

We also tested on two completely different languages, one low level (C++) and another high level (Python), to see if the expected cache behaviour is observed in both.

All the times are measured in ms.

5 Evaluation Methodologies

In order to eliminate as much of noise/variance in the tests we used a digital ocean droplet instance with a Intel(R) Xeon(R) CPU E5-2650 CPU with 8GB of ram, running Ubuntu 18.04 x86.

Each algorithm was ran 3 times and the times averaged.

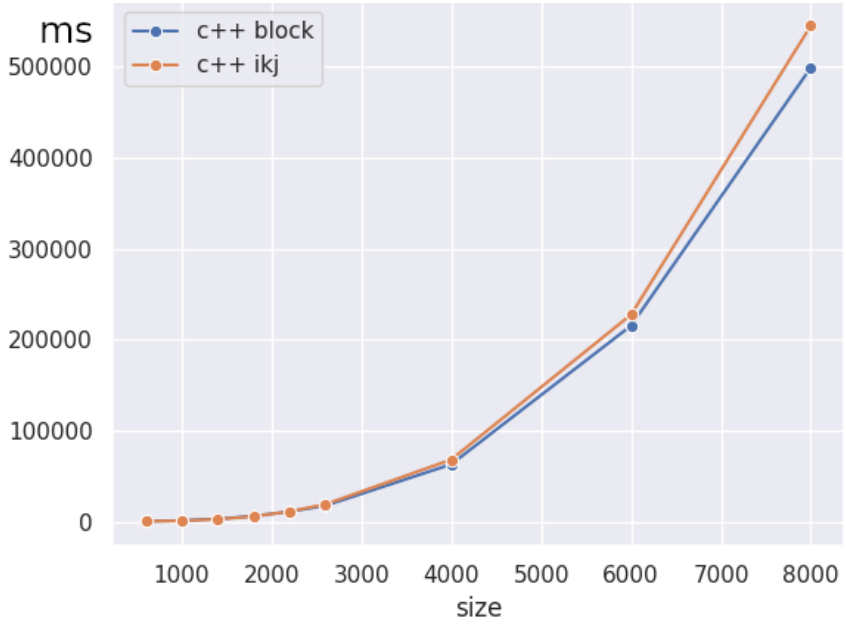
For C++ we used the STD library **Chrono** and for the Python implementations the **timeit** library. Both are known to provide accurate readings of time and the **timeit** library is set to only measure system time.

All tests, graphs, and outputs are controlled and generated using a script in python in order to provide consistency in the results.

To evaluate the percentage of cache misses for each algorithm we used **Valgrind** a well known library for program profiling.

Java and *Python* programming languages were used to implement the 1st and 2nd algorithms, both having similar behaviour.

6 Results and Analysis



As expected the first algorithm performs the worse, wasting a lot of CPU time on memory accesses due to the many cache misses.

```
I refs:         1,082,491,636
I1 misses:      1,752
LLi misses:     1,676
I1 miss rate:   0.00%
LLi miss rate:  0.00%

D refs:         405,964,347 (269,223,421 rd + 136,740,926 wr)
D1 misses:      134,649,213 (134,581,341 rd + 67,872 wr)
LLd misses:     74,945 ( 7,932 rd + 67,013 wr)
D1 miss rate:   33.2% ( 50.0% + 0.0% )
LLd miss rate:  0.0% ( 0.0% + 0.0% )
LL refs:        134,650,965 (134,583,093 rd + 67,872 wr)
LL misses:      76,621 ( 9,608 rd + 67,013 wr)
LL miss rate:   0.0% ( 0.0% + 0.0% )
```

The second algorithm fares much better due to its lower cache miss ratio.

```
I refs:         1,082,229,474
I1 misses:      1,753
LLi misses:     1,677
I1 miss rate:   0.00%
LLi miss rate:  0.00%

D refs:         539,919,928 (403,179,003 rd + 136,740,925 wr)
D1 misses:      16,923,818 ( 16,855,946 rd + 67,872 wr)
LLd misses:     74,945 ( 7,932 rd + 67,013 wr)
D1 miss rate:   3.1% ( 4.2% + 0.0% )
LLd miss rate:  0.0% ( 0.0% + 0.0% )

LL refs:        16,925,571 ( 16,857,699 rd + 67,872 wr)
LL misses:      76,622 ( 9,609 rd + 67,013 wr)
LL miss rate:   0.0% ( 0.0% + 0.0% )
```

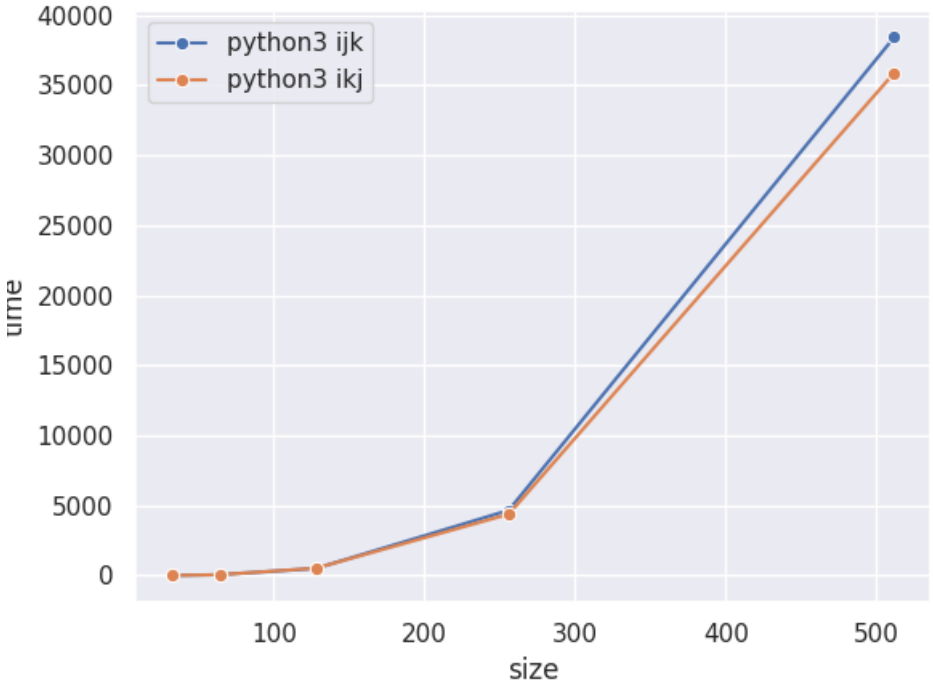
And finally the third gets the best performance with a lower miss ratio even for relative small matrices.

```
I refs:         6,633,402
I1 misses:      1,581
LLi misses:     1,533
I1 miss rate:   0.02%
LLi miss rate:  0.02%

D refs:         3,042,813 (521,653 rd + 2,521,160 wr)
D1 misses:      74,923 ( 7,910 rd + 67,013 wr)
D1 miss rate:   2.7% ( 2.6% + 2.7% )
LLd miss rate:  2.5% ( 1.5% + 2.7% )

LL refs:        83,230 ( 15,397 rd + 67,833 wr)
LL misses:      76,456 ( 9,443 rd + 67,013 wr)
LL miss rate:   0.8% ( 0.1% + 2.7% )
```

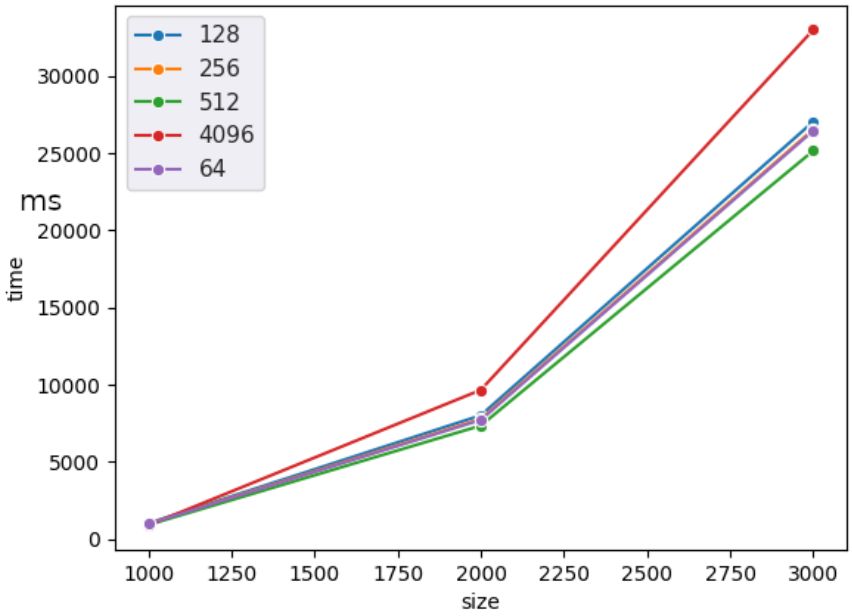
(LL is last level cache which corresponds to L3 cache)

Running the first and second algorithm in a completely different language such as **Python** shows similar results albeit less pronounced and like the other

ones the difference becomes more apparent with ever increasing size.



Lastly we tested what should be the size of the sub matrix block for the third algorithm.



7 Conclusions

From this paper we can conclude that memory access patterns can greatly influence the execution time of a program. A trivial change in something like a for loop can result in speedups of two or more times. Understanding the underlying architecture and hardware is a must for the optimization of a program.

In the context of matrix multiplication this is clearly shown in the three different algorithms that we tested. As we push the size of matrices higher and higher, it becomes apparent that the fastest one is loop tilling, a algorithm that subdivides each matrix into smaller ones, it achieves the lowest cache miss ratio and consequently is the fastest. This algorithm has a parameter, the size of the sub matrices, and from the testing we can conclude that it has to be a value small enough not to cause cache overflow.