U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Avaliação da *performance* de um *single core*

Trabalho realizado no âmbito da UC Computação Paralela e Distribuída, pelos alunos
André Pereira, up201905650
Margarida Vieira, up201907907
Matilde Oliveira, up201906954

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Descrição do Problema

O trabalho tinha como objetivo estudar os efeitos no desempenho do processador da hierarquia de memória aquando do acesso a quantidades significativamente grandes de dados.

Para tal, o foco assentou na multiplicação de matrizes, dado tratar-se de um sub-problema extremamente comum e utilizado na resolução de outros problemas maiores, desde a resolução de sistemas de equações lineares até ao treinamento de redes neuronais profundas.

Algoritmos

Para cumprir com o objetivo do trabalho, foram implementados três algoritmos diferentes de multiplicação de matrizes. A diferença entre estes algoritmos assenta essencialmente na forma como tiram partido da alocação em memória dos números, linhas e colunas das matrizes sendo que, para cada matriz, é inicialmente alocado um espaço de memória contínuo.

1. Multiplicação Simples de Matrizes (Versão 1)

O algoritmo de multiplicação mais convencional, segue aquela que é a definição matemática de multiplicação de matrizes, apresentada na figura 1.

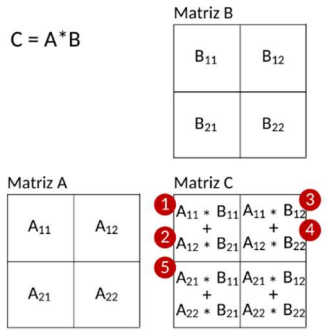

Figura 1

Cada elemento da matriz resultado corresponde à soma dos produtos dos elementos da sua linha na primeira matriz pelos elementos da sua coluna na segunda matriz.

Assim, o algoritmo implementado guarda os elementos da matriz A e matriz B num *array*, com os elementos guardados linha a linha em memória (figura 2). A matriz resultado C é guardada da mesma forma, quando calcula elemento a elemento.

Implementado na função *OnMult()* em C++ e *on_mult()* em Rust.

2. Multiplicação Simples de Matrizes (Versão 2)

A versão 2 do algoritmo de multiplicação simples toma partido da forma como os valores da matriz são guardados em memória.

A matriz A foi armazenada tal como na versão 1 – *row major*, mas a matriz B passou a ter os seus valores guardados por coluna – *column major* (figura 3).

Para realizar o cálculo da matriz resultado C, a metodologia é a mesma da versão 1. Cada valor da matriz C é calculado individualmente e guardado no respetivo local do array resultado. No entanto, ao iterar sobre a matriz B, os elementos relevantes para o cálculo encontram-se mais próximos na memória.

Implementado na função *OnMult2()* em C++ e *on_mult2()* em Rust.

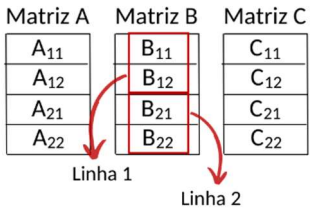

Figura 2

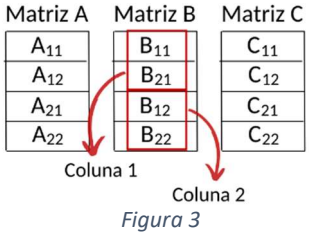

Figura 3

3. Multiplicação por Linha de Matrizes

O algoritmo de multiplicação por linha de matrizes consiste em multiplicar cada elemento da primeira matriz por cada um dos elementos da linha correspondente na segunda matriz.

Os elementos de todas as matrizes são guardados linha a linha – *row major*, contudo o cálculo do valor da matriz resultado C é realizado aos poucos, ou seja, a cada iteração o valor atualmente guardado na matriz é incrementado do resultado na nova multiplicação.

Implementado na função *OnMultLine()* em C++ e *on_mult_line()* em Rust.

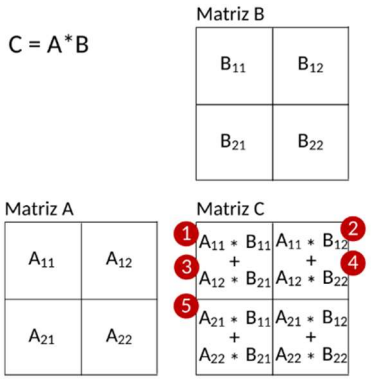

Figura 4

4. Multiplicação em Bloco de Matrizes

O algoritmo de multiplicação por blocos consiste em transformar a matriz original numa matriz composta por blocos, isto é, uma matriz cujos elementos matriciais são, eles próprios, sub-matrizes, com um certo tamanho previamente definido – às sub-matrizes dá-se, então, o nome de blocos e estes são encarados como meros elementos na matriz principal.

O cálculo da matriz resultado C obtem-se através da multiplicação pelos diferentes blocos. O resultado da multiplicação de cada bloco é parcialmente guardado na matriz final e, posteriormente, incrementado do resultado da multiplicação de blocos na mesma posição da matriz C.

O algoritmo utilizado para a multiplicação interna de um bloco por outro foi a multiplicação de matrizes em linha (explicada em 3. Multiplicação por Linha de Matrizes). Para além disso, e após se assumir um bloco como um elemento matricial, a multiplicação desses elementos é por sua vez também feita em multiplicação por linha.

Implementado na função *OnMultBlock()* em C++.

Métricas de Avaliação de Performance

Para analisar a *performance* dos algoritmos implementados, recorreu-se a várias métricas, sendo a mais intuitiva e óbvia o **tempo de execução** do programa na realização do cálculo da multiplicação de matrizes. Adicionalmente, no código em C++, utilizou-se a biblioteca PAPI – *Performance Application Programming Interface* – para medir o **número de falhas no acesso à memória cache de nível 1 e 2** e o **número de operações de vírgula flutuante**, FLOPs.

Foram executados vários testes, para diferentes tamanhos de matrizes (desde 600 x 600 até 10240 x 10240), com o objetivo de retirar uma amostra significativa para a compreensão do impacto dos diferentes algoritmos na *performance* do programa.

Para além disso, procedeu-se à implementação dos mesmos algoritmos numa outra linguagem de programação, **Rust**, com o objetivo de entender se tal acarretava algum impacto na *performance* do programa.

Resultados e Análise

Esta secção destina-se à análise dos valores retirados nos vários testes efetuados. Cada um dos algoritmos suprarreferidos foi executado três vezes para cada tamanho de matriz evidente no eixo horizontal dos gráficos. Portanto, os valores no eixo vertical dizem respeito à média dos valores obtidos nas três execuções efetuadas. De notar, no entanto, que tal não se aplica aos tamanhos de matriz superiores a 3000x3000, devido ao tempo de execução significativamente elevado.

Os testes foram realizamos com uma cache de dados nível 1 de tamanho 32KB e um cache unificada nível 2 de tamanho 256KB.

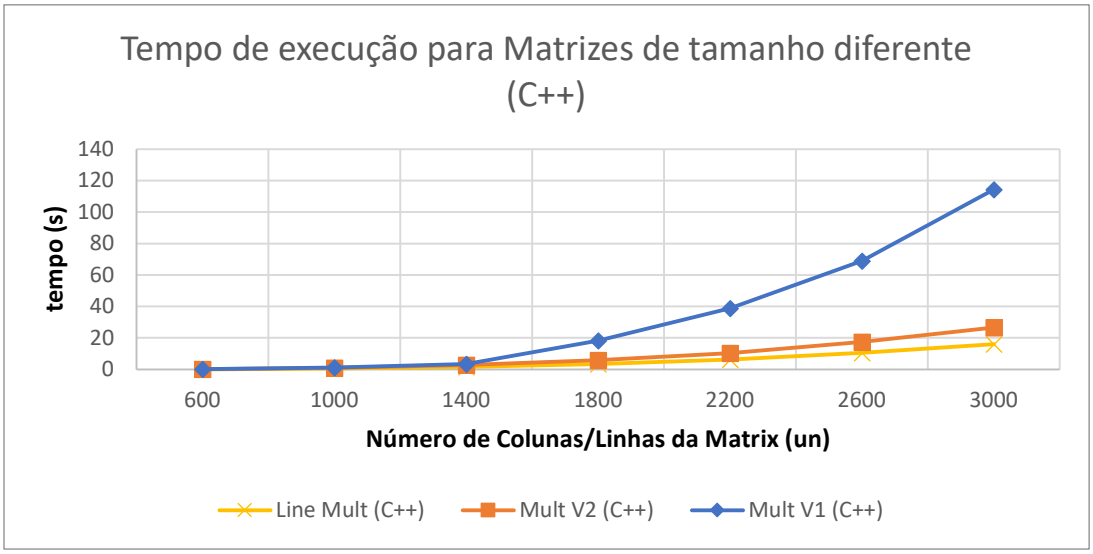

Figura 5

Através do gráfico acima (figura 5), podemos verificar que, independentemente do algoritmo, o tempo de execução da multiplicação de matrizes aumenta com o aumento do tamanho das mesmas. Já no tempo de execução, o algoritmo usado desempenha um papel de extrema relevância, principalmente para matrizes de tamanho elevado.

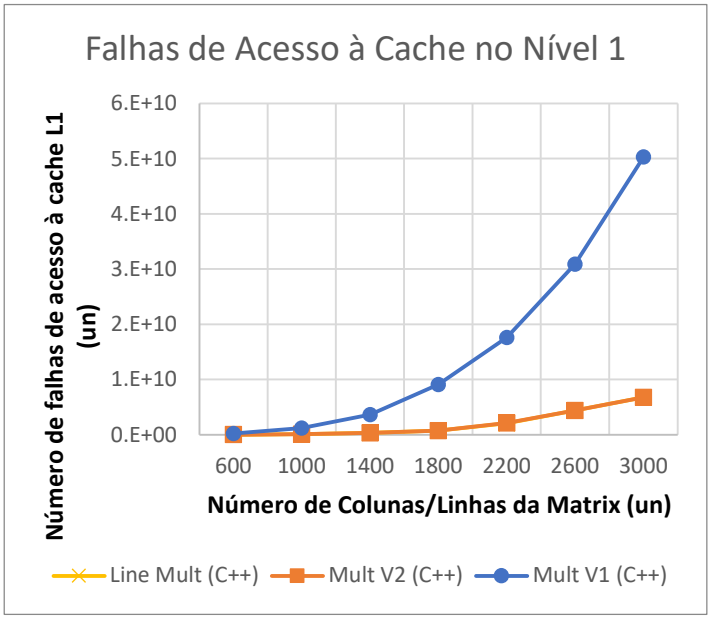

Figura 7

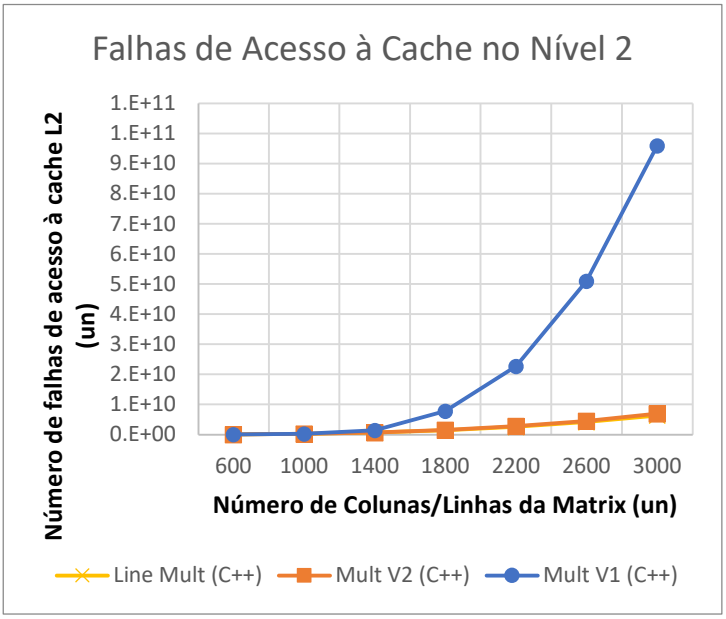

Figura 6

Os dois gráficos acima (figuras 6 e 7) tornam evidente a tendência para o aumento do número de falhas de acesso às caches nível 1 e 2 com o aumento do tamanho das matrizes. O *overhead* de acesso à memória inerente pode justificar o aumento dos tempos de execução.

Para além disso, é possível verificar que o número de falhas de acesso à cache nível 1 começa a aumentar em valores relativamente baixos do tamanho da matriz (1000x1000), em contraste com o número de falhas de acesso à cache nível 2, que apenas começa a aumentar em valores ligeiramente mais elevados (1800x1800). Isto deve-se ao tamanho das caches de nível 1 e 2, respetivamente 32 KB e 256 KB.

Ainda, independentemente do algoritmo usado, o número de falhas de acesso à cache nível 1 é menor que o número de falhas de acesso à cache nível 2 em tamanhos de matrizes diferentes (figuras 8 e 9).

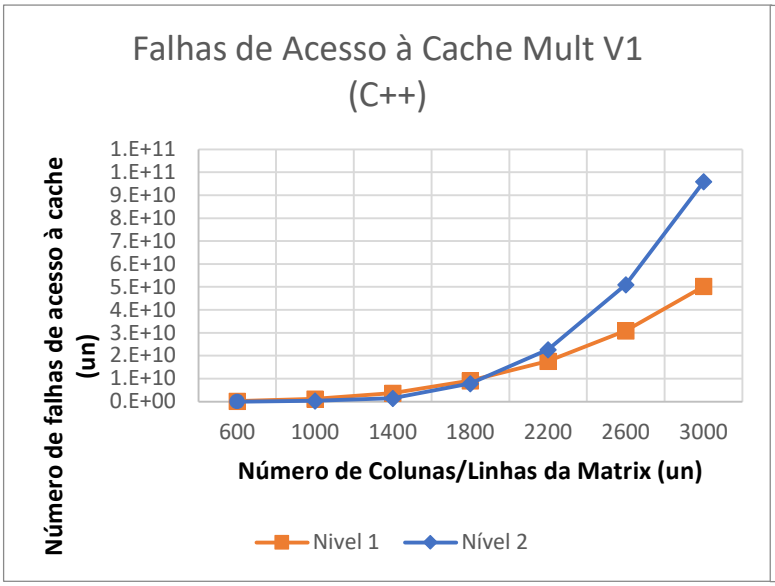

Figura 8

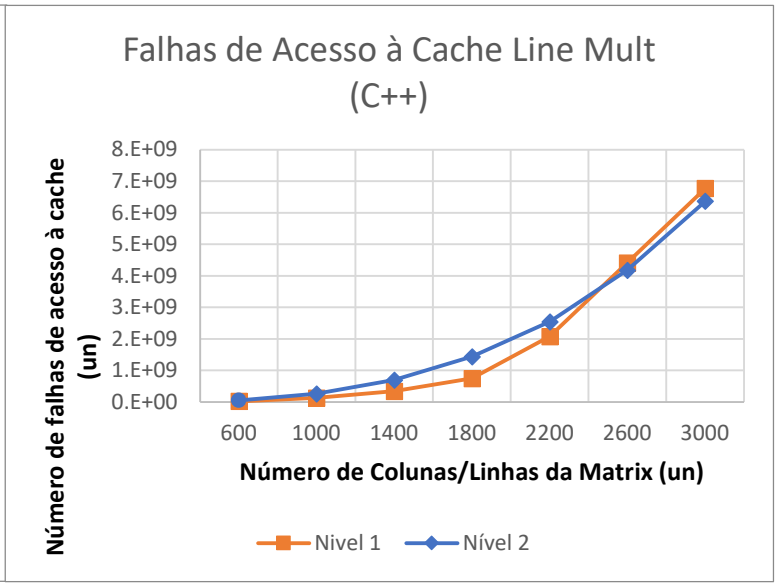

Figura 9

Na linguagem **Rust** nota-se uma tendência para o aumento do tempo da operação em análise. Contudo, algo que se torna evidente no gráfico ao lado (figura 10) é que o algoritmo *Line Mult* é mais demorado nesta linguagem do que o algoritmo *Mult V2*. Isto pode dever-se ao facto do algoritmo *Line Mult* ter necessidade de alterar o valor guardado na matriz resultado C várias vezes, em detrimento do algoritmo *Mult V2*, que calcula de imediato o valor a colocar na matriz resultado C, acedendo apenas uma única modificação.

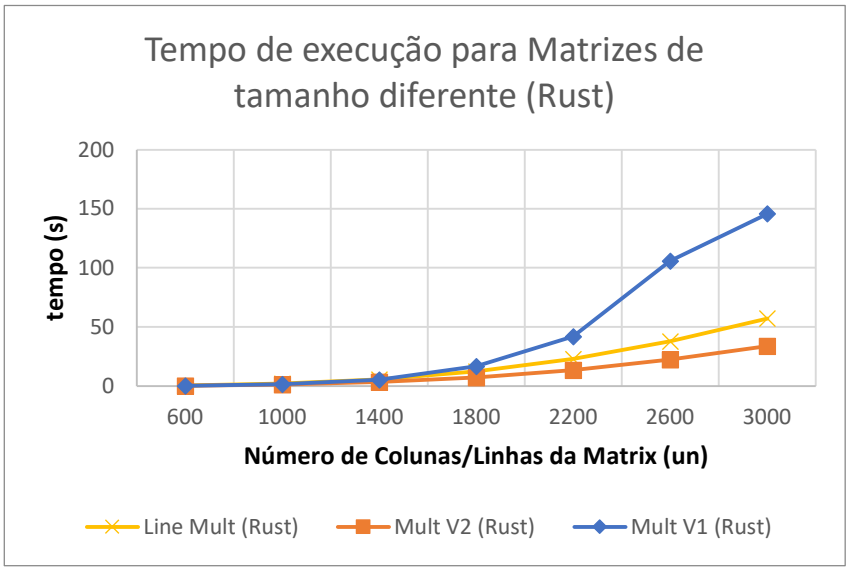

Figura 10

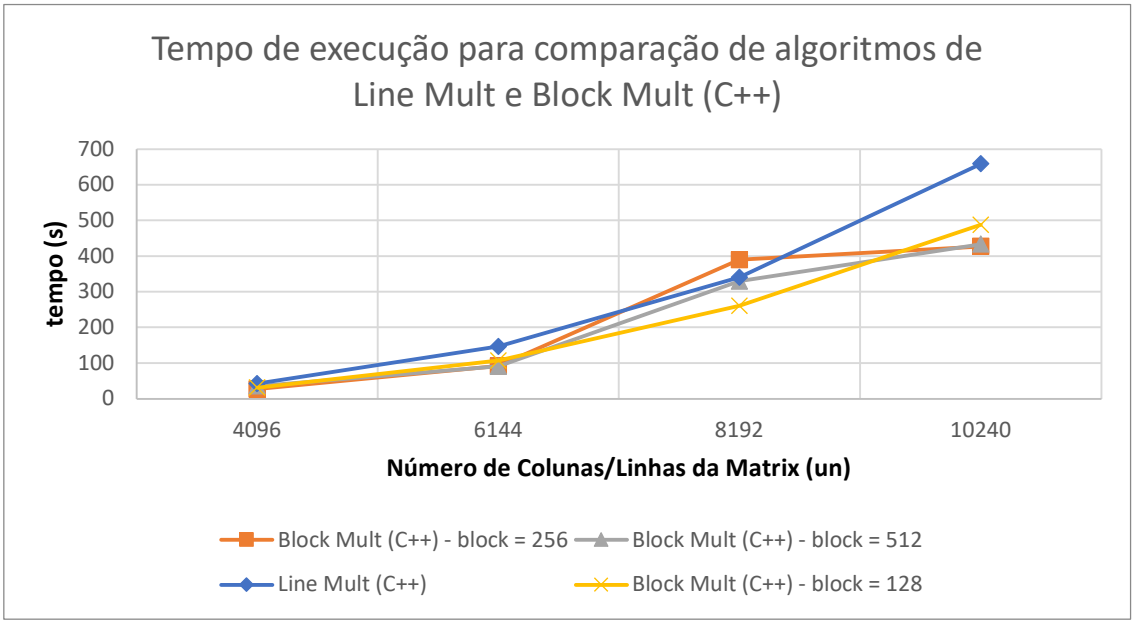

Figura 11

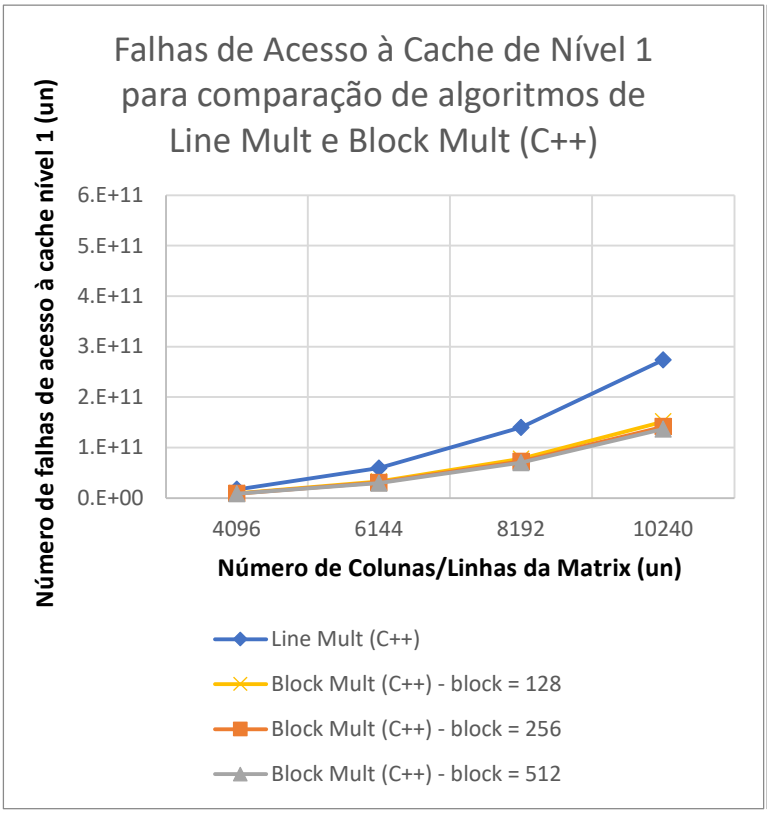

Figura 12

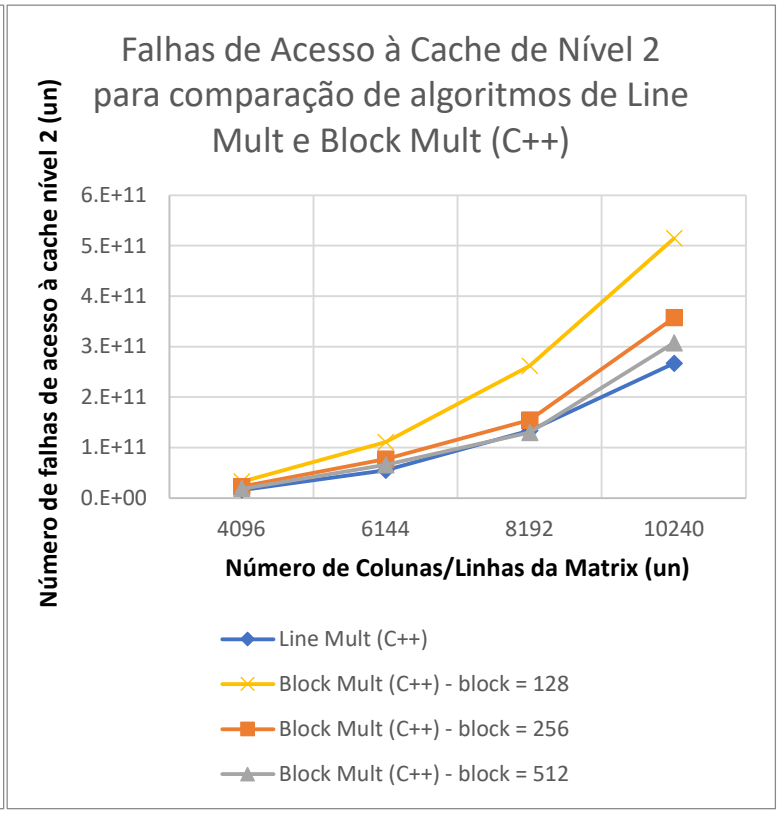

Figura 13

Na figura 11 podemos verificar que o algoritmo *Block Mult* é geralmente mais eficiente. Contudo, essa implementação tem leves variações relativamente ao tempo de execução para diferentes tamanhos de bloco, tornando-se isto mais evidente nos tamanhos de matriz 8192 e 10240. Ao analisar o número de falhas de acesso à cache nível 2 (figura 13), onde existe maior variação entre os algoritmos, verificamos que, ao usar um bloco de 128B, existe um maior número de falhas, o que, em teoria, levaria a um *overhead* no tempo de execução, o que não se verificou.

Uma possível explicação para eficiência do Block Mult com blocos de tamanho de 128B, assumindo que *miss* no nível 1 implica *hit* no nível 2, *miss* no nível 2 implica *hit* no nível 3, é que, ao existir vários *misses* no nível 2, existe, implicitamente, um maior número de *hits* no nível 3. Por outro lado, nos tamanhos de bloco de 256B e 512B não irão conseguir a maioria de informação na cache de nível 3.

Por fim, com o gráfico abaixo (figura 14), é possível concluir que o motivo do aumento do tempo de execução do cálculo da multiplicação matricial é o aumento do número de operações de vírgula flutuante realizadas, em consequência do aumento do tamanho das matrizes usadas.

Este dado permite-nos retirar, também, conclusões relativamente à capacidade do CPU que se prova ser constante, sendo esta calculada com o número de operações a dividir pelo tempo de execução da multiplicação de matrizes de diferentes tamanhos.

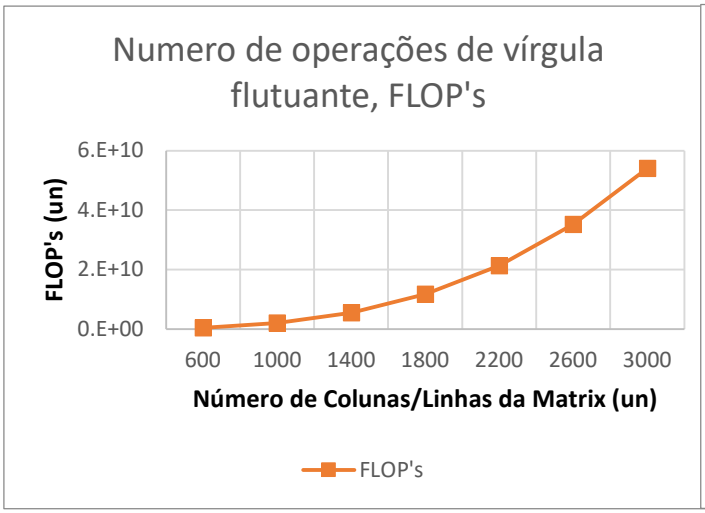

Figura 14

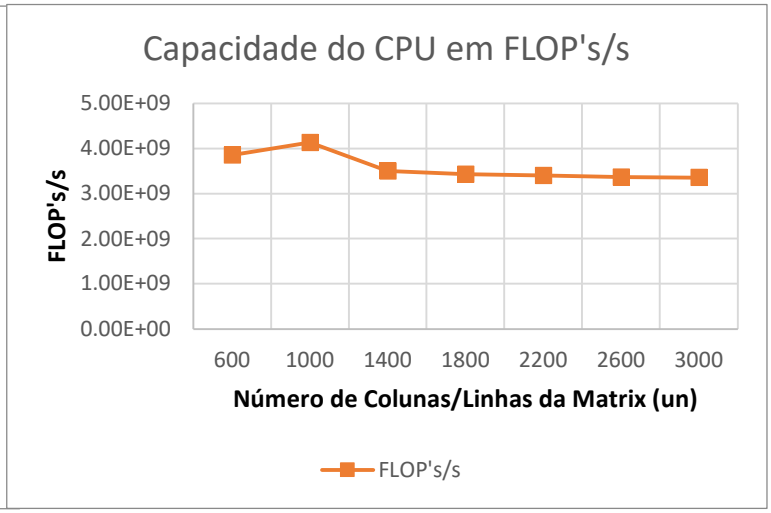

Figura 15

Conclusão

A realização deste primeiro trabalho permitiu-nos aprofundar os conceitos apresentados em aula relativos à gestão de memória. A partir da análise de resultados e da implementação de diferentes algoritmos, conseguimos tirar conclusões sobre o impacto que esta gestão e as decisões tomadas para alocação de espaços de memória pode ter na performance e nos tempos de execução de um programa tão custoso como a multiplicação de matrizes.

Assim, cumprindo os objetivos a que nos propusemos, conseguimos retirar conclusões e ganhar um melhor entendimento sobre esta matéria.