Performance Evaluation of a Single Core

1º Trabalho Laboratorial

Licenciatura em Engenharia Informática e Computação

Computação Paralela e distribuída

Turma 3 | Grupo 16:

Bruna Marques - up202007191

Francisca Guimarães – up202004229

Inês Oliveira – up202103343

2º Semestre

Ano Letivo 2022/2023

Índice

1. Introdução 2

2. Algoritmos 2

2.1 Algoritmo de Multiplicação Básico 2

2.2 Algoritmo de Multiplicação por Linha 2

2.3 Algoritmo de Multiplicação por Blocos 3

3. Métricas de Performance 3

4. Resultados 4

4.1 Comparação entre Algoritmos 4

4.2 Algoritmo de Multiplicação de Blocos com Blocos de Diferentes Tamanhos 4

4.3 Comparação entre Algoritmos Implementados em C/C++ e Java 5

4.4 Comparação entre Algoritmos e Performance de Cache 5

4.5 Comparação GFLOPS nos Algoritmos 6

5. Conclusão 6

1. Introdução

No âmbito da unidade curricular Computação Paralela e Distribuída, da Licenciatura em Engenharia Informática e Computação, foi proposto aos alunos o primeiro trabalho laboratorial, cujo objetivo consistiu em implementar e comparar o desempenho de três algoritmos de multiplicação de matrizes em duas linguagens distintas, nomeadamente C/C++ e Java, com vista a compreender os efeitos da utilização de grandes quantidades de dados na performance do processador. Para tal, foi ainda utilizada *a Performance Application Programming Interface (PAPI)*, de forma a recolher e analisar dados relevantes durante a execução do programa em C/C++.

2. Algoritmos

2.1 Algoritmo de Multiplicação Básico

O algoritmo em questão consiste na multiplicação de cada linha da primeira matriz pelas colunas da segunda matriz. Por outras palavras, é realizada uma iteração pelas linhas da primeira matriz, na qual, para cada linha, são percorridas as colunas da segunda matriz e é efetuado o cálculo da soma dos produtos dos elementos da linha e da coluna.

```
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;
    }
}
```

2.2 Algoritmo de Multiplicação por Linha

Neste algoritmo, embora semelhante ao anterior, é utilizada uma abordagem que consiste em multiplicar cada elemento de uma linha da primeira matriz pela linha correspondente da segunda matriz.

Uma vez que utilizamos *arrays* unidimensionais para representar matrizes, os elementos das matrizes pertencentes à mesma linha são guardados de forma consecutiva, isto é, ocupam espaços consecutivos em memória, enquanto que os elementos que pertencem à mesma coluna ou a matrizes diferentes são guardados separadamente.

Assim, e dado que com este algoritmo os elementos são percorridos por linha, de forma consecutiva, conseguimos minimizar os "data cache misses", o que se traduz num algoritmo melhorado, em comparação com o algoritmo anterior.

```
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];
                }
            }
        }
```

2.3 Algoritmo de Multiplicação por Blocos

No algoritmo anterior, é de salientar que se o tamanho das matrizes for suficientemente grande, uma só linha pode não caber na *cache*. Desta forma, para cada elemento da primeira matriz, iterar sobre a linha da segunda matriz, resultará em *cache misses,* uma vez que só uma parte da linha está armazenada na *cache*.

Assim, com o objetivo de corrigir esse problema, antes da multiplicação, este algoritmo divide as matrizes em blocos de tamanhos iguais. De seguida, para cada bloco, aplica-se o algoritmo de multiplicação por linha. No entanto, se o tamanho de cada bloco for muito grande, ocorre o mesmo problema acima mencionado.

```
for (int line_matrix_a = 0; line_matrix_a < num_blocks; line_matrix_a++) {
    for(int block_index=0; block_index < num_blocks; block_index++) {
        for(int col_matrix_b=0; col_matrix_b < num_blocks; col_matrix_b++) {
            int next_line_a = (line_matrix_a + 1) * bkSize;
            for(int i = line_matrix_a * bkSize; i < next_line_a; i++) {
                int next_block_a = (block_index + 1) * bkSize;
                for (int k = block_index * bkSize; k < next_block_a; k++) {
                    int next_block_b = (col_matrix_b+1)*bkSize;
                    for (int j = col_matrix_b*bkSize; j<next_block_b; j++) {
                        phc[i *m_ar+ j]+=pha[i*m_ar + k]*phb[k * m_ar + j];
                    }
                }
            }
        }
    }
}
```

3. Métricas de Performance

Para testar a performance dos algoritmos mencionados, utilizamos as métricas de tempo de execução e L1/L2 data cache misses. Para medir as métricas do hardware, utilizamos a *Performance Application Programming Interface (PAPI)*. Esta métrica é utilizada uma vez que através dos cache misses obtemos uma boa indicação de como a cache está a ser utilizada pelo algoritmo, o que tem um impacto na sua performance. Assim, quanto menor o número de cache misses, melhor a utilização da cache e maior a performance do algoritmo.

Para além de C/C++, os algoritmos de multiplicação básica e por linha foram também implementados em Java, registando os tempos de execução com o objetivo de comparar a performance e observar o impacto da escolha de linguagens de programação.

Para ambas as linguagens e para cada algoritmo estima-se a capacidade da máquina a partir do cálculo dos GFLOPS:

$$\text{FLOPS} = \frac{2\times(tamanho\ da\ matriz)^3}{tempo\ de\ execução}$$

4. Resultados

Após a recolha de medições, comparamos e analisamos os tempos de execução, o número de *cache misses* e os GFLOPS dos três algoritmos implementados. No terceiro algoritmo, também foi alvo de estudo os efeitos do tamanho do bloco.

Além disso, nos dois primeiros algoritmos, avaliamos o desempenho das duas linguagens de programação escolhidas, C/C++ e Java, em relação aos seus tempos de execução. Apresentamos, de seguida, a informação do hardware utilizado para estas medições:

CPU	Intel(R) Core(TM) i7-9700 CPU @3.00Hz			
CACHE	**TOTAL SIZE**	**LINE SIZE**	**NR. OF LINES**	**ASSOCIATIVITY**
L1 Data Cache	32 KB	64 B	512	8
L1 Instruction Cache	32 KB	64 B	512	8
L2 Unified Cache	256 KB	64 B	4096	4
L3 Unified Cache	12288 KB	64 B	196608	12

Figura 1 - Informações do Hardware usado

4.1 Comparação entre Algoritmos

Com o objetivo de comparar os algoritmos mencionados, medimos o tempo de execução de cada implementação em C/C++ para valores crescentes de tamanhos de matriz. Para o terceiro algoritmo, algoritmo de multiplicação em bloco, utilizamos o tempo médio dos tamanhos de bloco para cada dimensão de matriz. Como esperado, o algoritmo de multiplicação básico foi o mais lento entre os três, e o algoritmo de multiplicação por bloco um pouco mais rápido em comparação ao algoritmo de multiplicação por linha.

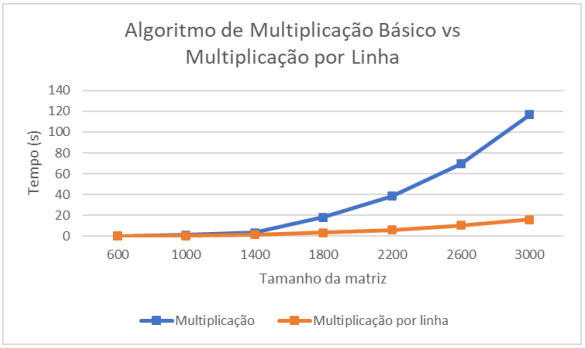

Figura 2 - Gráfico de comparação Multiplicação Básico vs Multiplicação por Linha em C++

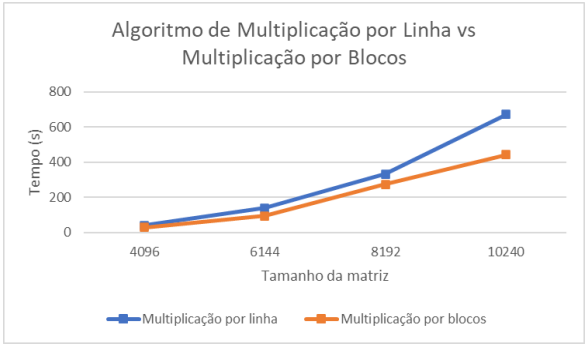

Figura 3 - Gráfico de comparação Multiplicação por Linha vs Multiplicação por Blocos em C++

4.2 Algoritmo de Multiplicação de Blocos com Blocos de Diferentes Tamanhos

Para avaliar o impacto do tamanho dos blocos no tempo de execução do algoritmo de multiplicação de blocos, utilizamos os tamanhos de bloco 128, 256 e 512, em matrizes com dimensões variando de 4096 até 10240, com um incremento de 2048 a cada iteração. Embora

tenhamos observado uma diferença de tempo relativamente pequena entre os resultados, não conseguimos tirar uma conclusão definitiva sobre como essa evolução afeta o desempenho do algoritmo. Assim, acreditamos que esta análise não permitiu chegar a uma conclusão clara sobre o impacto do tamanho dos blocos no tempo de execução do algoritmo.

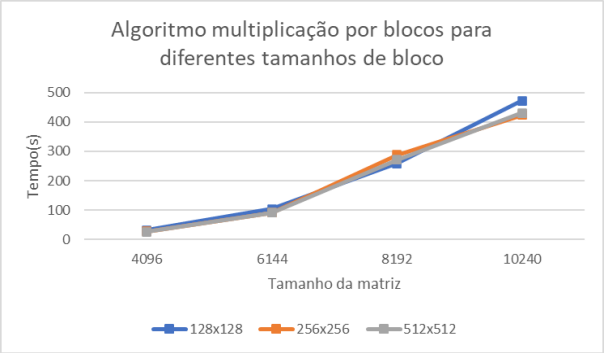

Figura 4 - Gráfico de comparação Multiplicação por Blocos para diferentes tamanhos de blocos

4.3 Comparação entre Algoritmos Implementados em C/C++ e Java

De modo a comparar o tempo de execução do mesmo algoritmo, em diferentes linguagens de programação, executamos o código em C/C++ e em Java para matrizes com valores crescentes de tamanhos. Após analisarmos os gráficos que seguem, concluímos que o programa em C/C++ teve um desempenho melhor que o de Java, melhorando os seus tempos consoante a velocidade do algoritmo. (Relembramos que o Algoritmo de Multiplicação Básico é mais lento que Algoritmo de Multiplicação em Linha).

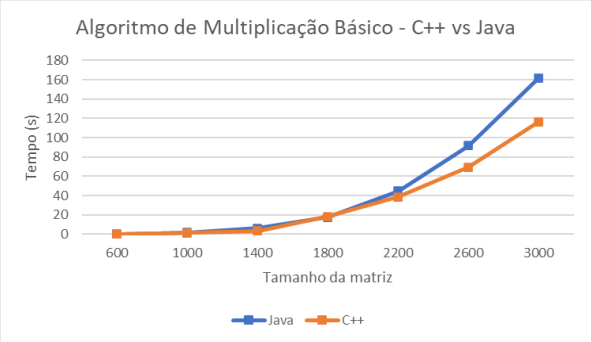

Figura 5 - Gráfico de comparação Algoritmo de Multiplicação Básico - C++ vs java

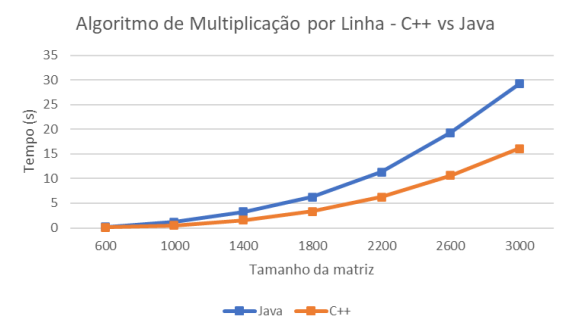

Figura 6 - Gráfico de comparação Algoritmo de Multiplicação por Linha - C++ vs Java

4.4 Comparação entre Algoritmos e Performance de Cache

Com o objetivo de analisar o impacto dos algoritmos e as suas otimizações na utilização da cache, todos os algoritmos foram caso de estudo para matrizes com valores incrementais. Conseguimos concluir, após a análise dos seguintes gráficos, que quando comparados os algoritmos de Multiplicação Básico e Multiplicação em Linha o número de *cache misses* é reduzido no segundo algoritmo nos dois níveis. Por outro lado, quando comparados os algoritmos de Multiplicação em Linha e Multiplicação por Blocos, o número de *cache misses* 1 é reduzido com o custo do aumento do número de *cache misses* 2.

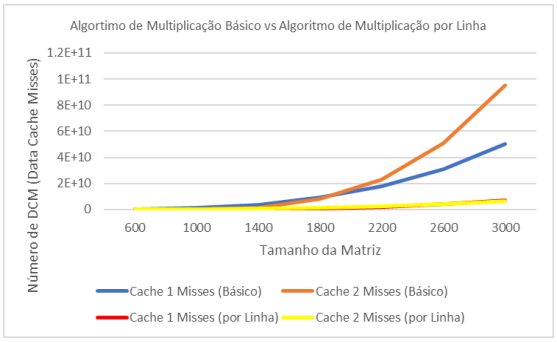

Figura 7 - Gráfico de comparação de *cache misses* entre Multiplicação e Multiplicação por linha

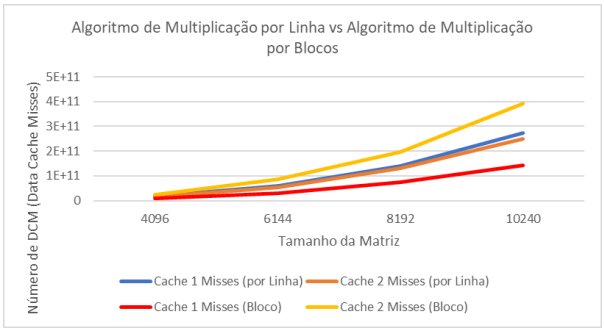

Figura 8 - Gráfico de comparação de *cache misses* entre Multiplicação por linha e Multiplicação por blocos

4.5 Comparação GFLOPS nos Algoritmos

Uma boa estratégia para verificar quão eficientes são os nossos algoritmos passa por calcular os GFLOPS, uma vez que fornecem uma medida quantitativa da velocidade de processamento de um determinado sistema. Assim, calculamos os GFLOPS para os algoritmos previamente mencionados, concluindo, como esperado, que os algoritmos implementados na linguagem C/C++ apresentam um melhor desempenho.

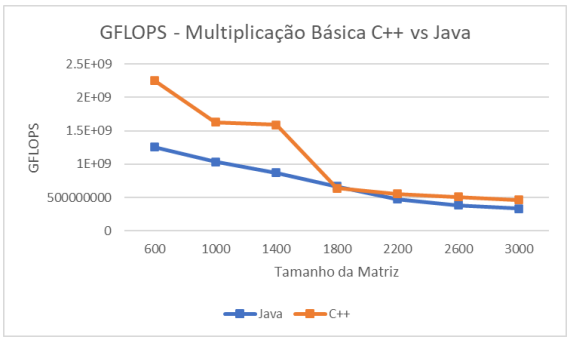

Figura 9 - Gráfico de comparação GFLOPS para Algoritmo de Multiplicação básica

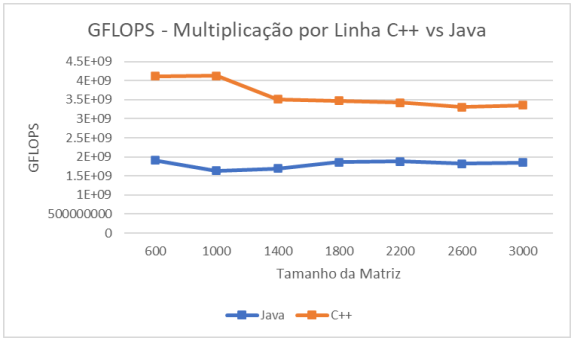

Figura 10 - Gráfico de comparação de GFLOPS Multiplicação por linha

5. Conclusão

Com este projeto, realizamos uma análise detalhada de como diferentes algoritmos de multiplicação de matrizes afetam o desempenho de um processador ao lidar com grandes conjuntos de dados, especialmente no que se refere à memória *cache*. Aprendemos a explorar a localidade espacial e temporal dos elementos armazenados em *cache*, o que nos permitiu reduzir significativamente o número de *cache misses* e, consequentemente, melhorar a velocidade e eficiência do programa.

Este trabalho ressalta a importância da otimização de algoritmos, um conceito fundamental que não deve ser subestimado ou negligenciado em qualquer projeto que envolva processamento de dados.