CPD project 1 - Part 1 - G17

Performance evaluation of a single core

1 - Objetivo:

Estudar o impacto do desempenho do processador na hierarquia de memória ao lidar com grandes volumes de dados.

2 - Descrição do problema:

De modo a alcançar o objetivo definido previamente, o produto de duas matrizes será utilizado para este estudo. O produto de matrizes é importante porque é uma operação fundamental em muitas áreas da ciência da computação, incluindo gráficos computacionais, *machine learning* e processamento de imagens.

Nesta parte do projeto é importante comparar as *performances* de três implementações diferentes do produto de matrizes, em duas linguagens distintas (C++ e Python). Para além disso, as matrizes estudadas necessitam de ter diversos tamanhos.

3 - Algoritmos implementados:

Para este projeto, foram implementados três algoritmos diferentes que visam medir o desempenho de um único núcleo (*single core*), quando exposto a uma grande quantidade de dados. Esses algoritmos são:

- Multiplicação Simples de Matrizes
- Multiplicação de Matriz por Linha
- Multiplicação de Matriz por Bloco

3.1 - Multiplicação Simples:

Este algoritmo foi fornecido pelos docentes, programado em C++. Multiplica duas matrizes, ou seja, multiplica uma linha da primeira matriz por cada coluna da segunda matriz. A complexidade do algoritmo é **O(n^3)**. Segue-se o *loop* encarregue de multiplicar os valores das matrizes:

```
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;
    }
}
```

Para além de C++, este algoritmo foi implementado em Python, seguindo a mesma lógica. Nas duas linguagens, registaram-se os valores para matrizes de 600x600 a 3000x3000, incrementando 400 ao número das linhas e das colunas.

3.2 - Multiplicação por Linha:

Para esta versão, implementou-se um algoritmo melhorado que multiplica um elemento da primeira matriz pela linha correspondente da segunda matriz. Este algoritmo, embora mantenha a complexidade O(n^3), é mais eficiente que o primeiro. Segue-se o *loop* correspondente:

```
for (i = 0; i < m_ar; i++)
{
    for (j = 0; j < m_br; j++)
    {
        for (k = 0; k < m_ar; k++)
        {
            phc[i * m_ar + k] += pha[i * m_ar + j] * phb[j * m_br + k];
        }
    }
}
```

Este algoritmo foi implementado tanto em C++ como em Python. Registaram-se os valores para matrizes de 600x600 a 3000x3000, incrementando 400 ao número das linhas e das colunas.

No caso do algoritmo em C++, registaram-se também os valores para matrizes de 4096x4096 a 10240x10240, incrementando 2048.

3.3 - Multiplicação por Bloco:

Finalmente, no que toca à multiplicação de matriz por bloco, foi implementada uma versão do algoritmo anterior que divide as matrizes em blocos mais pequenos, que são calculados separadamente e somados no final.

Embora a complexidade temporal permaneça igual, esta abordagem concentra-se na otimização do acesso aos dados na memória. Ao dividir as matrizes em blocos menores, mais dados podem ser armazenados com maior frequência em memórias de nível inferior e mais rápidas, como caches L1 e L2.

Assim, o terceiro e último algoritmo reduz substancialmente o tempo necessário para recuperar os dados das memórias de nível mais alto. Segue-se o *loop* correspondente:

```
for (i = 0; i < m_ar; i += bkSize)
{
    for (j = 0; j < m_br; j += bkSize)
    {
        for (k = 0; k < m_ar; k += bkSize)
        {
            for (i2 = i; i2 < min(i + bkSize, m_ar); i2++)
            {
                for (j2 = j; j2 < min(j + bkSize, m_br); j2++)
                {
                    for (k2 = k; k2 < min(k + bkSize, m_ar); k2++)
                    {
                        phc[i2 * m_ar + k2] += pha[i2 * m_ar + j2] * phb[j2 * m_br + k2];
                    }
                }
            }
        }
    }
}
```

Este algoritmo foi implementado em C++ e registaram-se os valores para matrizes de 4096x4096 a 10240x10240, incrementando 2048 ao número das linhas e das colunas. Foram utilizados blocos de 128x128, 256x256 e 512x512.

4 – Métricas de *Performance*:

Para avaliar o desempenho dos algoritmos nas versões C/C++, utilizou-se a API de Performance (PAPI), que proporciona acesso a várias métricas da CPU, incluindo os níveis de cache utilizados pelo processo. Além do tempo de execução do algoritmo, registou-se o número de Operações de Ponto Flutuante (FLOP) e o número de falhas de cache para L1 e L2. No caso das versões em Python, apenas se registou o tempo de execução.

De modo a garantir integridade dos dados registados, todas as medições foram realizadas no mesmo computador. Este computador estava a ser utilizado numa máquina virtual com o sistema operativo Linux 5.15.146.1-microsoft-standard-WSL2. O processador é um AMD Ryzen 7 7840HS, com uma frequência de relógio máxima de 4.9 GHz.

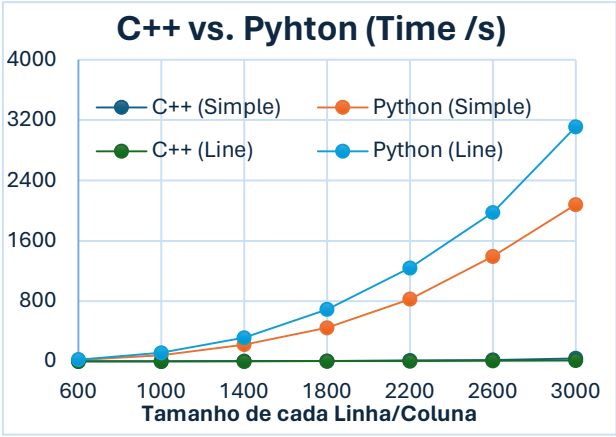
Na versão C/C++, compilou-se o programa utilizando a *flag* de otimização -O2, que aumenta o tempo de compilação e o desempenho do código gerado.

Para cada algoritmo implementado, tanto em C++ como em Python, foram registados os valores correspondentes 6 vezes. Desta forma, os dados registados assemelham-se, de certo modo, à realidade.
Nota: Registar os valores 6 vezes já foi difícil, pelo que mais do que isso seria inconcebível, devido ao tempo de execução de alguns dos algoritmos.

5 – Resultados e análise:

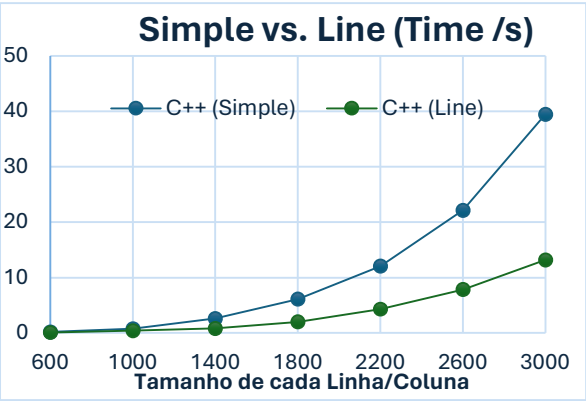
O registo dos dados pode ser encontrado no documento Excel presente no mesmo diretório que o atual documento. De aqui em diante, todos os dados utilizados correspondem ao valor médio das 6 medições.

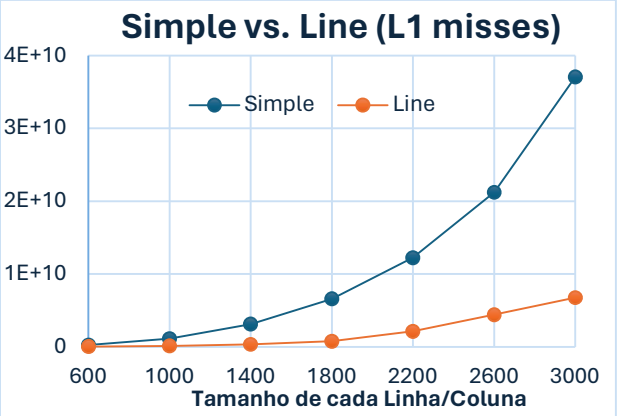
5.1 C++ vs. Python

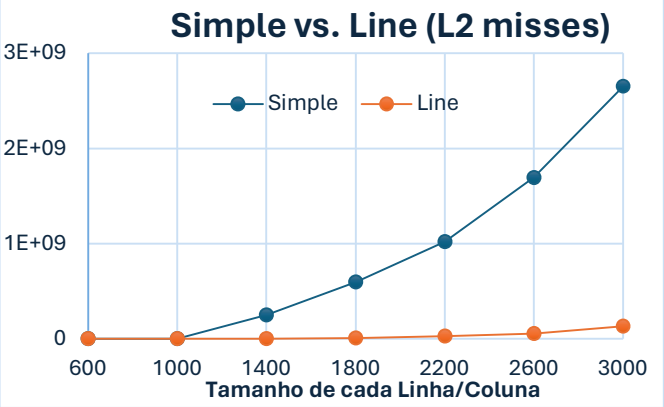


No que toca à comparação das duas linguagens, C++ é muito mais eficiente que o Python a executar os algoritmos, tal como se pode observar no gráfico. Isto acontece devido ao tipo de cada linguagem: C++ é uma linguagem compilada, onde é necessário compilar o código para se obter o executável, enquanto que o Python é uma linguagem interpretada (o código fonte é executado por um interpretador).

5.2 Multiplicação simples vs. Multiplicação por linha



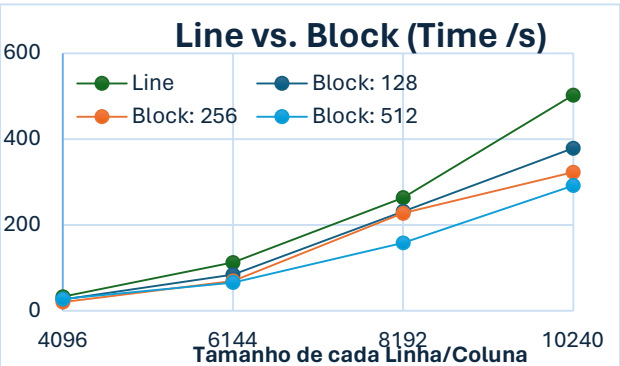


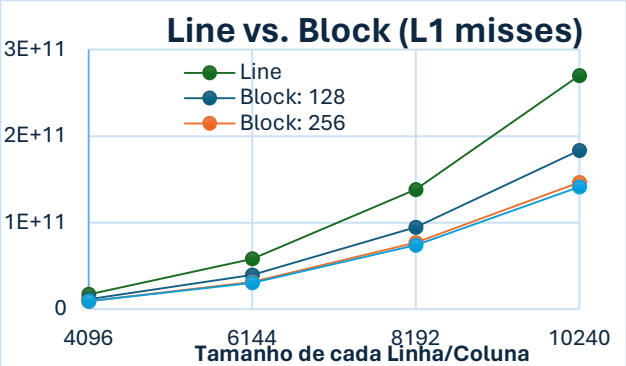


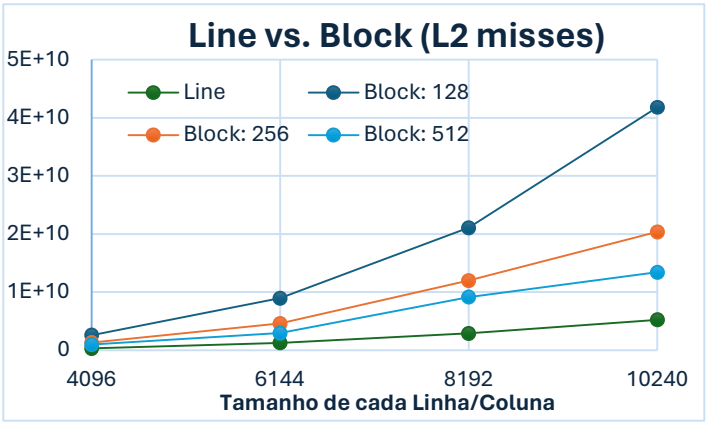
Ao analisar os gráficos destes dois algoritmos, torna-se evidente que o segundo algoritmo (*Line Multiplication*) supera significativamente o primeiro algoritmo (*Simple Multiplication*) em eficiência. Isto deve-se à capacidade do segundo algoritmo produzir os resultados mais rapidamente e com uma redução notável na perda de cache.

O segundo algoritmo é mais eficiente que o primeiro porque acede à memória de forma mais direta e organizada, evitando repetições que causam falhas de cache. Também não utiliza uma variável temporária: simplifica o cálculo acumulando os resultados diretamente na matriz de destino, reduzindo a quantidade de operações *write*.

5.3 Multiplicação por linha vs. Multiplicação por bloco







Depois de analisar os gráficos dos algoritmos *Line Multiplication* e *Block Multiplication*, pode-se observar que o segundo é mais rápido. Além disso, é importante destacar que quanto maior o tamanho do bloco, mais rápido é o processo.

Também é claro que as perdas de *cache* L1 são mais significativas para o primeiro algoritmo. Além disso, nota-se que a perda de *cache* diminui à medida que o tamanho do bloco aumenta. No entanto, em relação às perdas de *cache* L2, é possível observar que ocorrem menos no primeiro algoritmo. Por outro lado, as perdas de *cache* e são maiores quanto menor for o bloco (segundo algoritmo).

O primeiro algoritmo sofre mais com perdas de cache L1 devido ao acesso repetido e disperso à memória, resultando em mais falhas de cache L1. Por outro lado, as perdas de cache L2 são menores devido à capacidade e eficiência do cache L2 em lidar com mais dados e acessos. Para o segundo algoritmo, perdas de cache são maiores com blocos menores devido à fragmentação dos dados e mais acessos dispersos à memória, tanto em L1 quanto em L2.

CPD project 1 - Part 2 - G17

Performance evaluation of a multi-core

1 - Objetivo:

Estudar as diferenças de performance entre sistemas com um núcleo e sistemas com mais núcleos.

2 - Descrição do problema:

De modo a comparar ambos, sujeitar-se-ão diferentes implementações de sistemas com pluridade de núcleos aos testes realizados anteriormente.

3 – Algoritmos Implementados

Na segunda parte do projeto, para além do algoritmo de multiplicação por linha implementado previamente, utilizaram-se dois novos algoritmos (ambos utilizam multiplicação por linha):

3.1 – Alternativa 1:

```
#pragma omp parallel for private(i, j, k)
for (i = 0; i < m_ar; i++)
{
    for (j = 0; j < m_br; j++)
    {
        for (k = 0; k < m_ar; k++)
        {
            phc[i * m_ar + k] += pha[i * m_ar + j] * phb[j * m_br + k];
        }
    }
}
```

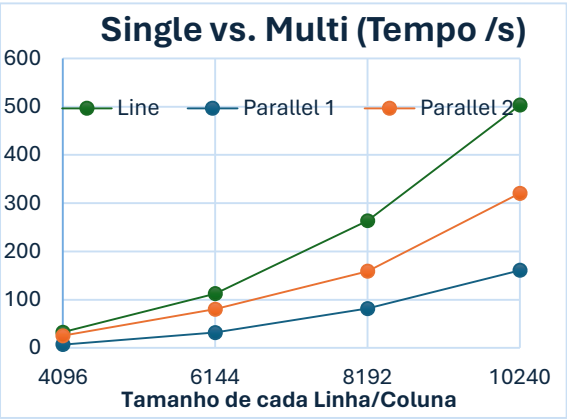
A primeira linha indica que o *loop for* seguinte é paralelizado. Cada *thread* tem as suas próprias cópias das variáveis i, j e k, de modo a garantir que as iterações do *loop* interno não interferem umas com as outras.

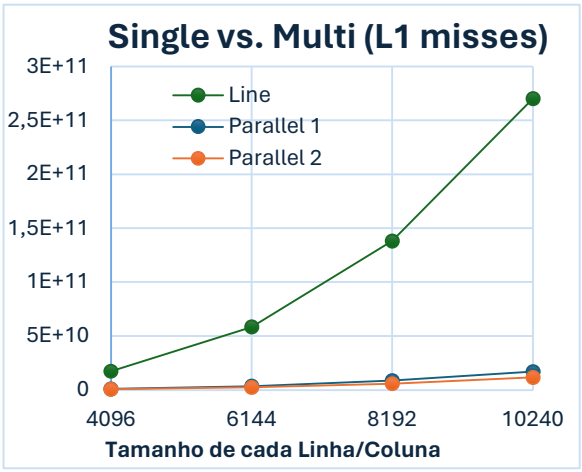
3.2 - Alternativa 2:

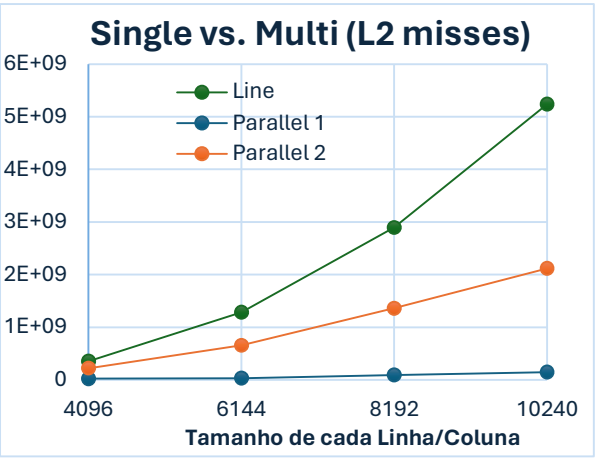
```
#pragma omp parallel private(i, j)
for (i = 0; i < m_ar; i++)
{
    for (j = 0; j < m_br; j++)
    {
        #pragma omp for
        for (k = 0; k < m_ar; k++)
        {
            phc[i * m_ar + k] += pha[i * m_ar + j] * phb[j * m_br + k];
        }
    }
}
```

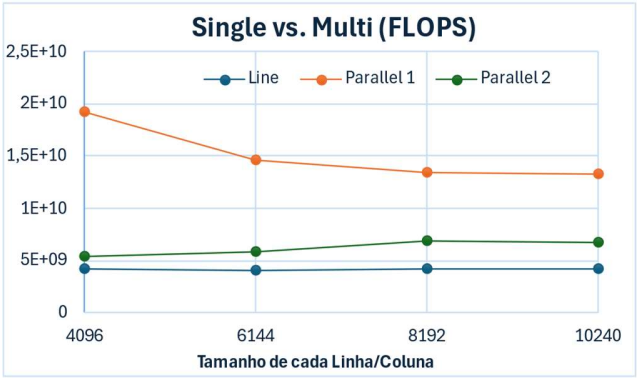
Desta vez, a primeira linha cria um grupo de *threads* e '#pragma omp for' paraleliza o *loop* mais interno (*for* k). Cada *thread* executa uma parte, o que equilibra a carga.

4 – Resultados e Análise









Como é possível observar nos gráficos, o tempo de execução do código no modo single core em relação ao modo multiple core é muito maior, tornando a paralelização mais eficiente em tempo que a computação sequencial. Isto deve-se ao facto de ser possível dividir entre os vários "CPU cores" do computador, podendo realizar as tarefas necessárias à computação de forma paralela, salvando tempo de computação.

Em relação a cache misses L1 e L2, vemos também que a computação paralela tem menos "Cache Misses" que a computação sequencial. Isso acontece, porque, ao utilizarmos só um processador para executar o programa na computação sequencial, acabamos por ter um maior conflito pelo espaço de cache do CPU, o que origina a que mesmos espaços de memória acabem por tentar ser utilizados (daí o conceito de Cache Miss). Utilizando todas as CPUs, por sua vez, estamos abertos a um maior mapeamento de cache para cada Core, o que origina menos cache misses por execução da multiplicação matricial.

Em relação ao número de FLOPS, consegue-se observar que a computação Single Core apresenta menor número que a programação Multi Core. Tal deve-se ao facto de que, precisamente por ser mais rápido temporalmente, e dado que FLOPS é "Floating-Point Operations per Second", e dado que o número de operações para a mesma matriz de tamanho N é igual, então, sendo a taxa temporal inversamente proporcional ao número de FLOPS, a programação paralela terá maior número de FLOPS (confirmando o resultado acima).

Por fim, para calcular o "Speedup" e a Eficiência da computação paralela vs. sequencial utilizamos as seguintes fórmulas:

$$Speedup = \frac{T_{seq}}{T_{paralel}};\ Efficiency = \frac{Speedup}{\#Cores}$$

Então, obtemos os seguintes resultados:

Algoritmos com Multi Core	Speedup	Efficiency
Algoritmo 1	1,477496584	18,47%
Algoritmo 2	3,613727631	45,17%

Conclusões

Em retrospetiva, a realização da primeira parte deste trabalho permitiu aprimorar os nossos conhecimentos quanto à gestão de memória, no que toca a melhorar a eficiência do programa. Mesmo sem o uso de computação paralela, conclui-se que é possível melhorar programas sequenciais manipulando a memória a nosso favor.

Relativamente à segunda parte, conclui-se que apesar de ser possível melhorar a eficiência de outras maneiras, a paralelização é, sem dúvida melhor que as estudadas anteriormente no projeto.

Finalmente, conclui-se que as melhorias no tempo de execução entre algoritmos demonstram o quão crítico é planear a implementação do código para reutilizar o máximo de memória de nível inferior (por exemplo, caches L1 e L2) e não desperdiçar tempo essencial em memórias de nível superior e latência.