Computação Paralela e Distribuída

Relatório 1: Multiplicação de Matrizes

António Ribeiro up201906761
Diogo Maia up201804974
Filipe Pinto up201907747

Explicação dos algoritmos

Algoritmo 1: Multiplicação genérica/ Schoolbook's algorithm

A multiplicação genérica segue o algoritmo feito "à mão":

1. Fixamos uma linha da $\mathbf{M}_{esq}$.
2. Iteramos cada elemento da $\mathbf{M}_{esq}$, multiplicando-o com o elemento correspondente da coluna selecionada $\mathbf{M}_{dir}$ ($\mathbf{M}_{dir}^{[i,k]}$ * $\mathbf{M}_{dir}^{[k,j]}$), acumulando esse mesmo valor numa variável temporária.
3. Continuamos esta iteração pela linha e coluna de cada uma das matrizes, ao terminarmos o loop de k, temos na variável temporária a célula correspondente *i, j* da matriz resultado, e podemos prosseguir para uma nova coluna, mantendo a linha.
4. Repetimos este processo para todas as linhas da $\mathbf{M}_{esq}$.

Iteração para o cálculo da primeira célula do resultado (cores representam multiplicações):

$$\begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \cdot \begin{bmatrix} b_0 & b_1 & b_2 \\ b_3 & b_4 & b_5 \\ b_6 & b_7 & b_8 \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_1b_3 + a_2b_6 & \dots & \dots \\ & \dots\dots & \\ & \dots\dots & \end{bmatrix}$$

Implementação em python:

```
for i in range(0, m_ar): # iteração linha a linha matriz esq
   for j in range(0, m_br): # iteração coluna a coluna da matriz esq
     temp = 0 # acumulador do valor da célula resultado
     for k in range(0, m_ar): # iteração da coluna
       temp += m_a[i * m_ar + k] * m_b[k * m_br + j]
     m_c[i * m_ar + j] = temp # posicionar o resultado na matriz
```

Algoritmo 2: Multiplicação por linha

A multiplicação comporta-se da seguinte forma:

1. Fixamos uma linha da $\mathbf{M}_{esq}$ (seguindo o algoritmo convencional acima).
2. Em vez de saltarmos para a linha seguinte mantendo a coluna (como faríamos "à mão"), mantemos a multiplicação de cada elemento da $\mathbf{M}_{esq}$ (selecionado na iteração) com uma linha inteira da $\mathbf{M}_{dir}$, colocando-os no respectivo lugar da matriz de resultado.
3. A cada posicionamento deste tipo, acedemos ao valor já acumulado na célula do resultado, para que cada vez que mudamos de linha (da $\mathbf{M}_{esq}$) possamos somá-lo com a nova multiplicação.

Como será explicado na análise de resultados, esta implementação é, em teoria, mais vantajosa dada a implementação da hierarquia de memória num determinado computador.

Iteração sob a primeira linha, para o cálculo da primeira linha do resultado:

$$\begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \cdot \begin{bmatrix} b_0 & b_1 & b_2 \\ b_3 & b_4 & b_5 \\ b_6 & b_7 & b_8 \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_1b_3 + a_2b_6 & a_0b_1 + a_1b_4 + a_2b_7 & a_0b_2 + a_1b_5 + a_2b_8 \\ & \dots\dots\dots\dots\dots\dots & \\ & \dots\dots\dots\dots\dots\dots & \end{bmatrix}$$

Implementação em python, com a alteração que inverteu os *loops k* e *j* interiores:

```
for i in range(0, m_ar):# iterador de acesso aos elementos das linha da matriz esquerda
  for k in range(0, m_ar):# fixação da linha horizontal da matriz direita
    for j in range(0, m_br):# acesso horizontal às colunas da matriz direita
        m_c[i*m_ar+j] += m_a[i*m_ar+k] * m_b[k*m_br+j]
```

Algoritmo 3: Multiplicação por bloco

A multiplicação por bloco comporta-se da seguinte forma:

1. Subdividir ambas as matrizes em blocos *B* de tamanho igual, gerando blocos de matrizes, abstraídos a “elementos” de uma outra matriz *M’*.
2. Aplicamos o algoritmo mais eficiente que conhecemos (*Alg. 2*, multiplicação por linha), tanto para a seleção dos blocos da matriz *M’*, como também para o cálculo individual da multiplicação de cada *B*.

Subdivisão em blocos:

$$\left[\begin{array}{cc|cc} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ \hline a_8 & a_9 & a_{10} & a_{11} \\ a_{12} & a_{13} & a_{14} & a_{15} \end{array}\right] \cdot \left[\begin{array}{cc|cc} b_0 & b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 & b_7 \\ \hline b_8 & b_9 & b_{10} & b_{11} \\ b_{12} & b_{13} & b_{14} & b_{15} \end{array}\right] \equiv \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \cdot \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} = \begin{bmatrix} A_0B_0 + A_1B_2 & A_0B_1 + A_1B_3 \\ A_2B_0 + A_3B_2 & A_2B_1 + A_3B_3 \end{bmatrix}$$

Block selection and matrix multiplication done with the line multiplication algorithm

Implementação em python:

```
for ii in range(0, m_ar, blkSize): # iteração sob a linha de blocos da Mesq
   for kk in range(0, m_br, blkSize): # iteração sob a linha de blocos da Mdir
       for jj in range(0, m_br, blkSize): # iteração sob as colunas de blocos da linha escolhida da Mdir
         # algoritmo previamente descrito, começando a iteração na secção do bloco
          for i in range(ii, ii + blkSize):
            for k in range(kk, kk + blkSize):
               for j in range(jj, jj + blkSize):
                    matrixC[i * m_ar + j] += m_a[i*m_ar+k] * m_b[k*m_br+j]
```

Explicação das métricas de avaliação de performance

Especificações do sistema usado:

- Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, 8 Cores, 2 Threads/Core, x86_64.
- L1d: 128 KiB, L2: 1MiB, L3: 8 MiB;
- RAM: 7887160 kB ;
- Execução feita com o mínimo de processos em background (OS: Linux Mint);

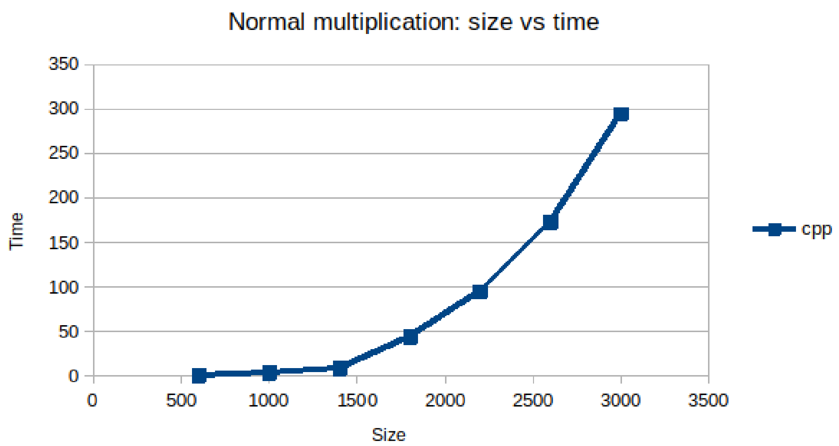
Métricas:

- **Tempo de execução:** métrica mais comum e simples de quantificar, para avaliação de sistemas. Quanto maior é esta quantia, mais tempo recursos estão indisponíveis para outras ações, mais tarde temos os resultados pretendidos e mais energia é gasta no cálculo da multiplicação das matrizes. Devemos procurar minimizá-lo.
- ***GFLops/s*:** medição da quantidade de operações realizadas ao longo do cálculo por segundo. Para isso, podemos recorrer à fórmula $2 * n^3 / \Delta t$ (nº máximo de operações de multiplicação e adição dividido pelo tempo de execução). Cálculos deste tipo são operações custosas (tendo em conta a mantissa, expoente e alocação de registos *FP*), o que poderá ser um fator determinante da *performance*.
- **Falhas/*Misses* de leitura da *cache* (nível 1 e nível 2, *PAPI_L1_DCM, PAPI_L2_DCM*):** Um dos objetivos do trabalho é estabelecer o impacto da má gestão de memória nos diferentes algoritmos. Este *overhead* é introduzido cada vez que os dados que procuramos não estão na região de memória mais próxima e rápida. Por este motivo, é essencial perceber que escolha traz consigo mais situações em que o acesso a *cache* não é aproveitado, e em que os níveis de memória mais custosos têm de ser utilizados.

Resultados e Análise

Algoritmo 1: Multiplicação genérica

O gráfico seguinte representa a evolução do tempo de execução com o crescimento do tamanho das matrizes em pleno CPP:



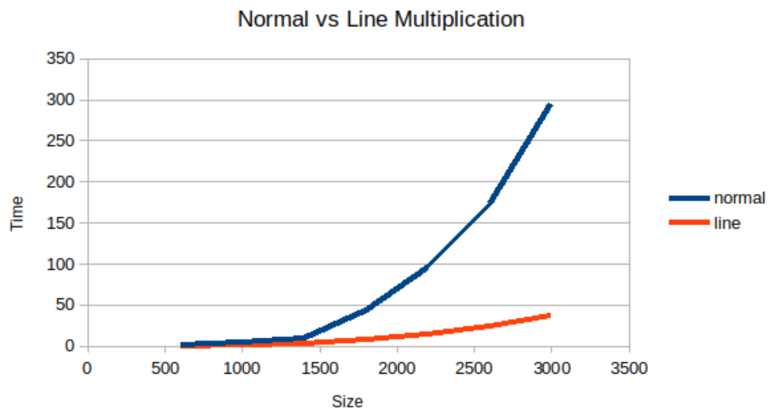
Python	
Size	Time
600	75.60906
1000	380.0178
1400	1089.823
1800	2333.325
2200	4295.777
2600	7134.925
3000	11106.5

Como podemos ver no gráfico, estamos perante uma curva exponencial, o que faz sentido, sendo que estamos a fazer a multiplicação de duas matrizes nxn (sendo *n* o número de valores numa coluna) e o número de operações cresce de forma exponencial com o tamanho. Quanto a falhas de leitura da cache, ambos seguem o mesmo crescimento exponencial aqui visto.

Quanto aos resultados em python, obtivemos uma curva semelhante, no entanto a ordem de grandeza dos valores de tempo obtidos são consideravelmente maiores devido às técnicas *built-in* do *python* quando se trata de operações em memória. *Python* é também uma linguagem de *scripting* interpretada, onde se realiza grande parte do trabalho em segundo plano, como a inferência de tipos e gestão de memória, contribuindo com um *overhead* significativo, em comparação com uma linguagem já compilada e otimizada para código máquina (no pior caso, 5 min em *c++* e 185 min em *python*).

Algoritmo 2: Multiplicação por linha

O gráfico seguinte representa a evolução do tempo de execução com o crescimento do tamanho das matrizes em, pleno CPP e a sua comparação com a multiplicação genérica:



Python

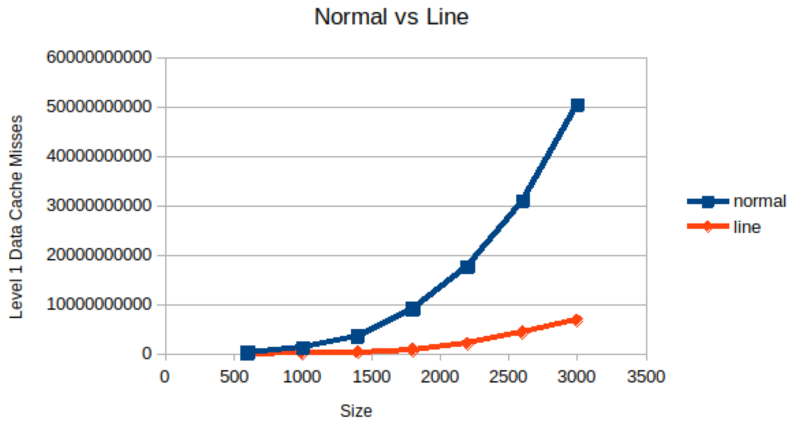
Size	Time
600	103.894
1000	485.409
1400	1346.39
1800	2876.76
2200	5289.61
2600	9205.54
3000	13769

Como podemos ver, estamos outra vez perante curva exponencial, no entanto, a taxa de variação da multiplicação em linha é bastante menor do que a multiplicação genérica. Isto pode-se explicar pela redução da necessidade de substituição de memória em cache, levando a uma redução do overhead na multiplicação em linha.

Sabemos que o acesso à *cache* é originado por dois fatores (<u>temporal e probabilístico</u>). Utilizando-os a nosso favor, extrai-se da memória RAM a quantidade máxima de dados que poderemos necessitar para a memória cache, que devido à forma como o *array* está alocado, serão os valores imediatamente a seguir à célula do segundo operando na multiplicação (parte ou toda a linha da $\mathbf{M}_{dir}$).

Desta forma, apesar de uma célula da matriz resultado não ser calculada de uma só vez, existe uma maior hipótese de encontrar os elementos da linha a ser multiplicados em cache (aumentando a performance, devido ao menor custo de *fetch*).

Isto pode ser comprovado perante os valores de falhas de informação na cache de nível 1 no gráfico seguinte.



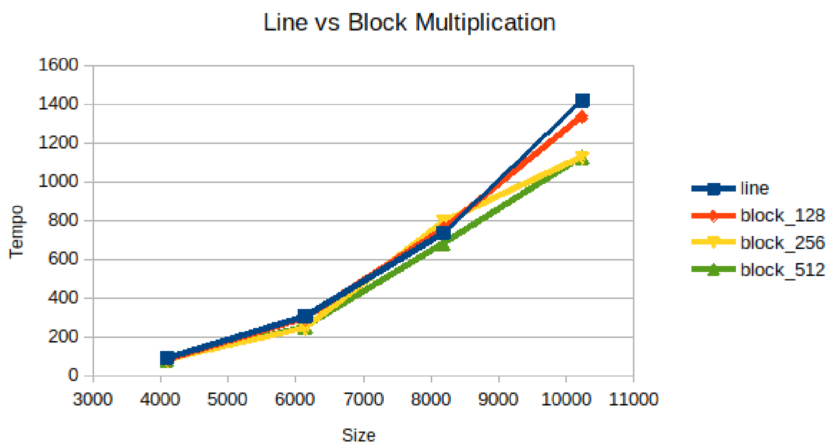
Quanto aos dados obtidos em *python*, podemos verificar o mesmo que na multiplicação normal.

Possíveis motivos o pior desempenho do algoritmo em python:

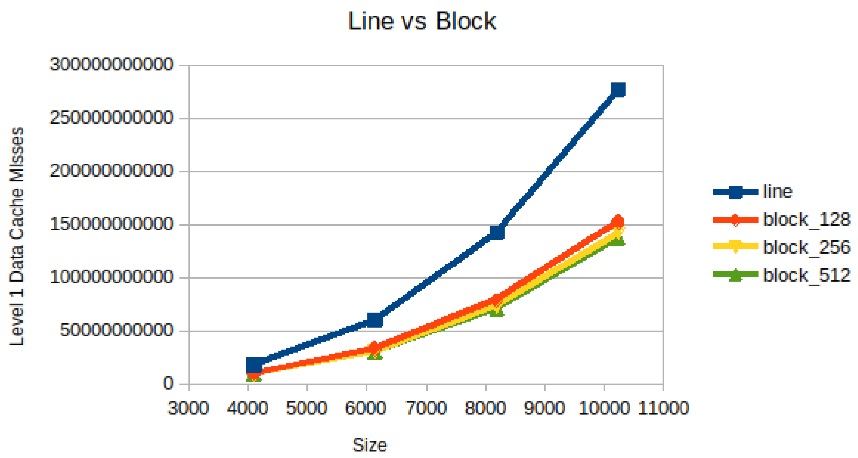
- No primeiro algoritmo usamos uma variável temporária para acumular o valor, neste usamos a célula da matriz resultado (podendo acrescentar *overhead*, dado o custo das chamadas internas de memória).
- Como FLOPs não são comutativas e associativas (dada o limite de precisão do tipo), a linguagem pode não realizar os *loops* e os acessos da forma esperada, e sim seguir uma sequência que lhe garante o resultado mais preciso em *runtime*.
- O armazenamento de uma estrutura do tipo *list* pode não ser o expectável (bloco de dados, mas sim linkedlist ou lista de referências), não sendo possível otimizá-lo da mesma forma.
- *"It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap." [Documentação oficial]*

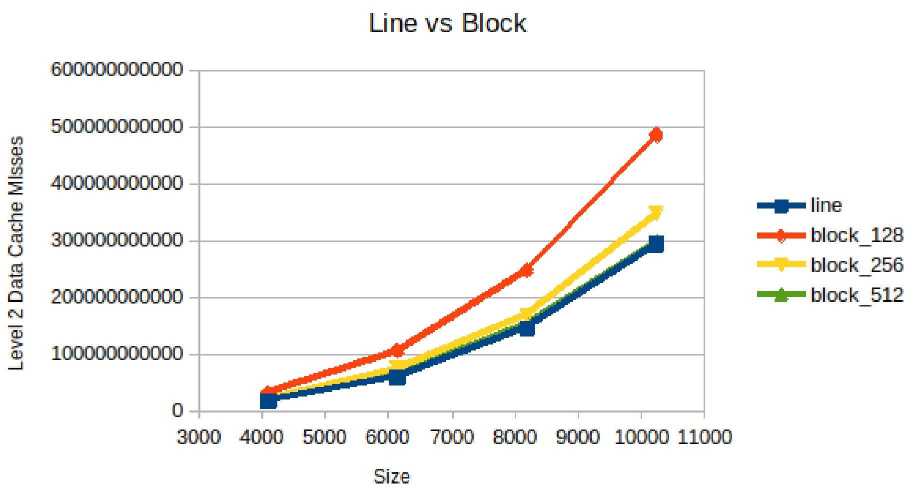
Algoritmo 3: Multiplicação por bloco

O gráfico seguinte compara o tempo de execução da multiplicação em linha e a multiplicação em bloco (tamanhos 128, 256 e 512):



Analisando o tempo, podemos ver que a multiplicação por bloco permite uma melhor performance em relação ao tempo de execução (sendo a melhor quando o bloco é de tamanho 512). No entanto, a diferença temporal entre as variações testadas não é elevada. O particionamento em blocos (com tamanhos adequados à hierarquia de memória) favorece o acesso à cache, dado que tendo as sub-matrizes a menor distância de CPU, é possível retirar o conjunto dados com menos custo de tempo para níveis cada vez mais baixos de memória.



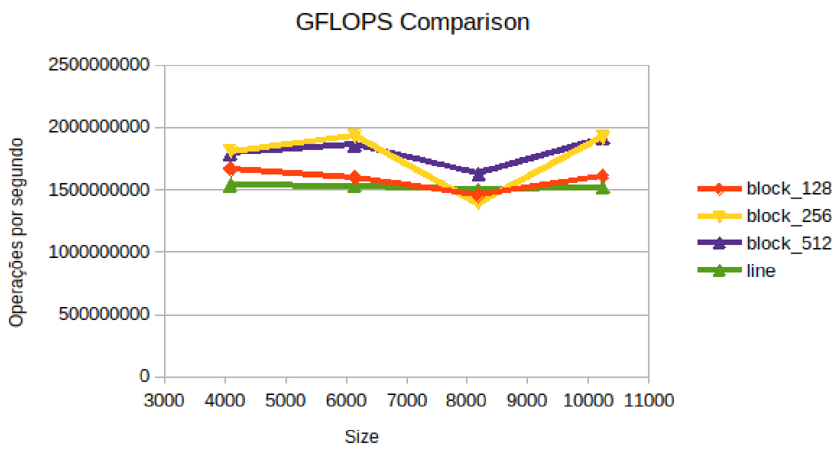


Mas, quando falamos de falhas de informação na cache de nível 1, o crescimento da quantidade de falhas é bastante mais elevado na multiplicação em linha do que nas multiplicações por blocos.

Em relação a L2, observamos que o algoritmo de bloco com 128x128 elementos (de 8 B) tem um pior desempenho que o algoritmo de linha. Isto poderá resultar da elevada limitação do número de elementos que são colocados em cache.

Em *L1* temos um espaço reduzido (128 KiB), que é preenchido na totalidade pelo bloco (125 KiB), o que torna direto o acesso à matriz neste caso.
Para *L2*, o mesmo bloco ocupa 13% do espaço, não sendo aproveitado, levando a mais *misses*. Blocos com maior tamanho (512x512) e o algoritmo de linha (com gestão implícita de memória) acabam por tirar mais proveito deste nível da cache. O tamanho de cada bloco tem de ser por isso equilibrado com os tamanhos disponíveis de *cache*.

Quando falamos em termos de GFLOPS, podemos reparar que independentemente do tamanho dos blocos e se a multiplicação é em linha ou em blocos, os valores tendem a estabilizar-se entre as 1500000000 e as 2000000000 operações por segundo (apesar de mudar o tamanho dos blocos, a matriz resultado é constante).



Conclusão

A análise dos resultados apresentados, leva-nos a concluir que para termos uma melhoria de performance, não basta analisar o algoritmo no sentido teórico. É necessário ter em conta todos os passos que um computador (numa arquitectura Von Neumann) tem de realizar, seja uma instrução de cálculo ou acesso à memória.

Do primeiro para o segundo algoritmo percebemos a importância que uma pequena, e quase imperceptível alteração (inversão da ordem dos loops), tem no desempenho temporal da multiplicação (em *C++*). A partir dessa observação, o algoritmo de divisão em blocos, é um passo lógico que progride esta cadeia de pensamento.

Percebemos também a importância da escolha da linguagem para problemas de elevada complexidade matemática, neste caso *C++* seria uma opção lógica para a realização dos cálculos (apesar de python poder ser “pseudo” compilado, como é o caso de *numba* com *JIT*).

Futuras melhorias deste algoritmo poderiam recorrer a novos métodos numéricos (como Coppersmith/Winograd), ou à adoção de threading/multicores (maximizando o aproveitamento da(s) unidade(s) de processamento).

Referências

Documentação Papi

- http://icl.cs.utk.edu/papi/docs/
- http://icl.cs.utk.edu/projects/papi-2.1/files/html_man/papi_presets.html

Algoritmos

- https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication
- https://iitd-plos.github.io/col729/lec/matrix_multiplication.html
- https://handwiki.org/wiki/Coppersmith%E2%80%93Winograd_algorithm
- https://dl.acm.org/doi/abs/10.1145/2213977.2214056?casa_token=OswrkZO3YaYAAAAA:BNrh4Pimq61n9zxC6Sf2lbJRNODK5uT2by9QtFCurpAAQN9JsvO01FWF1AV3WS-I8G5RHWN9GPIEm5w