COMPUTAÇÃO PARALELA E DISTRIBUÍDA

Análise de Performance do Processador no acesso a grandes quantidades de dados em algoritmos em Python e C++

Deborah Marques Lago - 201806102
Flávio Lobo Vaz - 201509918
José António Dantas Macedo - 201705226

Porto
2021/2022

1. Introdução

a. Contextualização

No contexto da unidade curricular Computação Paralela e Distribuída, o presente projeto tem como objetivo analisar a performance de processadores durante o acesso a grandes quantidades de dados, sendo neste caso feita a análise através de diferentes algoritmos de multiplicação de matrizes. A comparação será feita entre códigos nas linguagens de programação C++ e Python, com o recurso de uma API (PAPI) para recolha de métricas e posterior análise de resultados.

b. Descrição do Problema

Antes de equacionarmos a utilização como por exemplo da programação paralela, sabendo que a performance de um algoritmo depende de fatores tais como: velocidade de entrada/saída (I/O), padrão de acesso a dados, hierarquia de memória, entre outros. É proeminente primeiro analisar do ponto de vista da otimização do código, algumas das considerações explanadas nos seguintes exemplos :

1- Se o código em questão tem uma arquitetura bem estruturada e dessa forma obter melhor performance dos recursos de hardware e software disponíveis .

2- Se está bem desenhado, utiliza os melhores conceitos matemáticos e da computação existentes, tendo sempre em conta o contexto e objetivos dos mesmos.

3- Se a arquitetura dos processadores e da cache são utilizados da melhor forma. Algo que inclusive será fundamental neste relatório.

Não podendo nós descorar o papel extremamente importante do compilador, variando consoante a linguagem de programação, este já realiza otimizações que advêm de uma série de técnicas utilizadas nos processadores modernos, de forma a aumentar a performance, tais como :

1- Cache

2- Parallelism

3- Pipelining

Contudo, neste trabalho iremos nos focar no padrão de acesso a dados (Cache) que é exprimido pelo programador por via do seu código. Algo que os compiladores muito dificilmente conseguem tratar ou alcançar.

c. Algoritmos

Em relação aos algoritmos utilizados para demonstrar os benefícios da otimização de código, irão incidir na multiplicação de matrizes, onde será possível aumentar a dimensão das mesmas para tamanhos suficientemente grandes para serem relevantes na análise.

Os tempos de processamento serão medidos por via de um programa implementado por nós, que fará a multiplicação de matrizes para diversas dimensões, que gravará os tempos e informação, entre outras variáveis pertinentes para a análise.

Os algoritmos de multiplicação de matrizes implementados tanto em C++ quanto em Python neste trabalho são:

1-Multiplicação de matrizes pelo método algébrico simples

O produto de matrizes é dado pela expressão:

Seja A uma matriz $m \times n$ e B uma matriz $n \times p$ então o seu produto C é uma matriz $m \times p$.

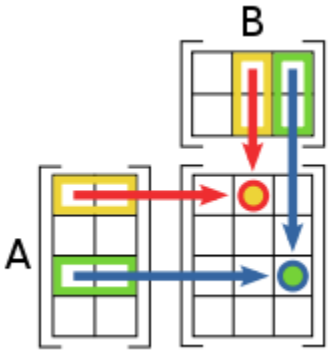
$$C = (AB)_{ij} = \sum_{r-1}^{n} a_{ir} b_{rj} = a_{i1} b_{1i} + a_{i2} b_{2i} + \ldots + a_{in} b_{nj}$$

para cada par *i* e *j* com $1 \leq i \leq m$ e $1 \leq j \leq p$.

O número de colunas da primeira matriz tem de ser igual ao número de linhas da segunda matriz, sendo que o produto de matrizes não é em geral comutativo, logo a ordem interessa.

O nosso algoritmo aplica diretamente a definição dada em cima . Assumiremos essa mesma definição como válida para os restantes algoritmos.

Exemplo:



$$(AB)_{1,2} = \sum_{r=1}^{2} a_{1,r} b_{r,2} = a_{1,1} b_{1,2} + a_{1,2} b_{2,2}$$

$$(AB)_{3,3} = \sum_{r=1}^{2} a_{3,r} b_{r,3} = a_{3,1} b_{1,3} + a_{3,2} b_{2,3}$$

Implementação :

```
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-Multiplicação linha a linha (1ª otimização)

Algoritmo para a multiplicação de matrizes mais eficiente em relação à multiplicação algébrica simples anterior. Com este algoritmo pretendemos

essencialmente aproveitar melhor a arquitetura e forma de funcionamento do processador, como de seguida iremos ver com mais detalhe.

Cada **elemento** da primeira matriz é multiplicado por cada elemento da **linha** correspondente da segunda matriz, segundo as regras do produto de matrizes.

Exemplo:

Implementação:

```
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_ar+k];
        }
    }
}
```

3-Multiplicação linha a linha mas utilizando sub-matrizes (blocos) mais pequenas, das matrizes (2ª otimização).

Neste algoritmo dividimos as matrizes em blocos (sub-matrizes) de tamanho igual, tendo em conta as características e regras definidas para o produto de matrizes no primeiro algoritmo . Desta forma, efetuamos para cada par de blocos correspondentes nas duas matrizes a multiplicação linha a linha como no algoritmo anterior.

Exemplo:

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 2 & 7 \\ 1 & 5 & 6 & 2 \\ \hline 3 & 3 & 4 & 5 \\ 3 & 3 & 6 & 7 \end{array}\right] \quad <=> \quad \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \qquad B = \left[\begin{array}{cc|cc} 3 & 1 & 1 & 4 \\ 7 & 4 & 5 & 3 \\ \hline 2 & 4 & 2 & 4 \\ 3 & 1 & 8 & 5 \end{array}\right] \quad <=> \quad \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

The block multiplication is performed the same way as before, but now the elements to multiply are matrices:

$$C = A * B = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} A11*B11+ A12*B21 & A11*B12+ A12*B22 \\ A21*B11+ A22*B21 & A21*B12+ A22*B22 \end{bmatrix}$$

Implementação:

```
// loops for matrix blocks multiplication
for(int i=0; i< m_ar; i+=bkSize){
    for(int j=0; j<m_br; j+=bkSize){
        for(int k=0; k<m_ar; k+= bkSize){

            //where we do the multiplications using blocks and we use the method on OnMult
            for(int ii=i; ii<i+bkSize; ii++){
                for(int jj=j; jj<j+bkSize; jj++){
                    for(int kk=k; kk<k+bkSize; kk++){
                        phc[ii*m_ar+kk] += pha[i*m_ar+jj] * phb[jj*m_ar+kk];
                    }
                }
            }
        }
    }
}
```

2. Métricas de Performance

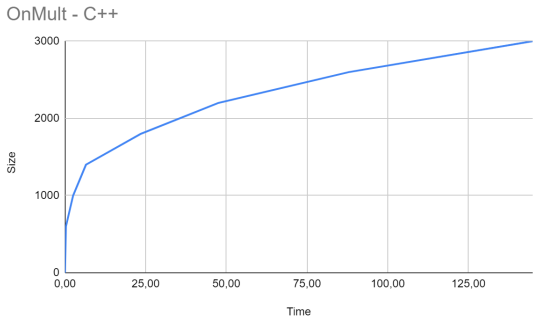
Utilizamos a Performance API para avaliação de performance do processador e dentre os eventos disponíveis, escolhemos mais especificamente eventos relacionados à medição do acesso à cache no decorrer da execução dos algoritmos, assim como a quantidade total de instruções realizadas.

```
PAPI_L1_DCM :
PAPI_L2_DCM
PAPI_L1_ICM
PAPI_L2_ICM
PAPI_L1_TCM
PAPI_L2_TCM
PAPI_TOT_INS
```

3. Resultados e Análise

Algoritmo 1 - OnMult

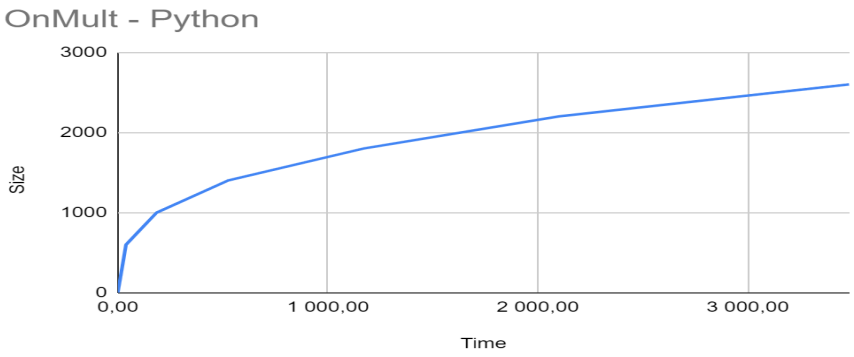
C++



-

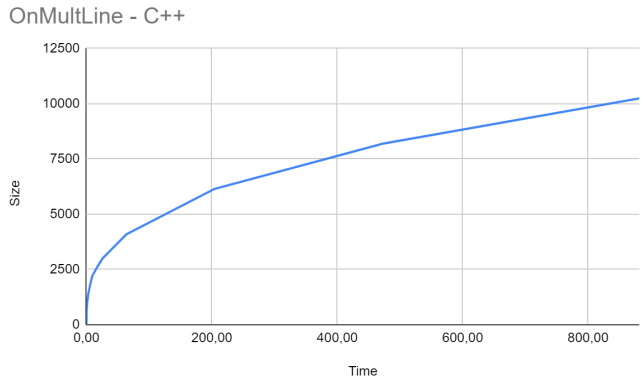
Python

-

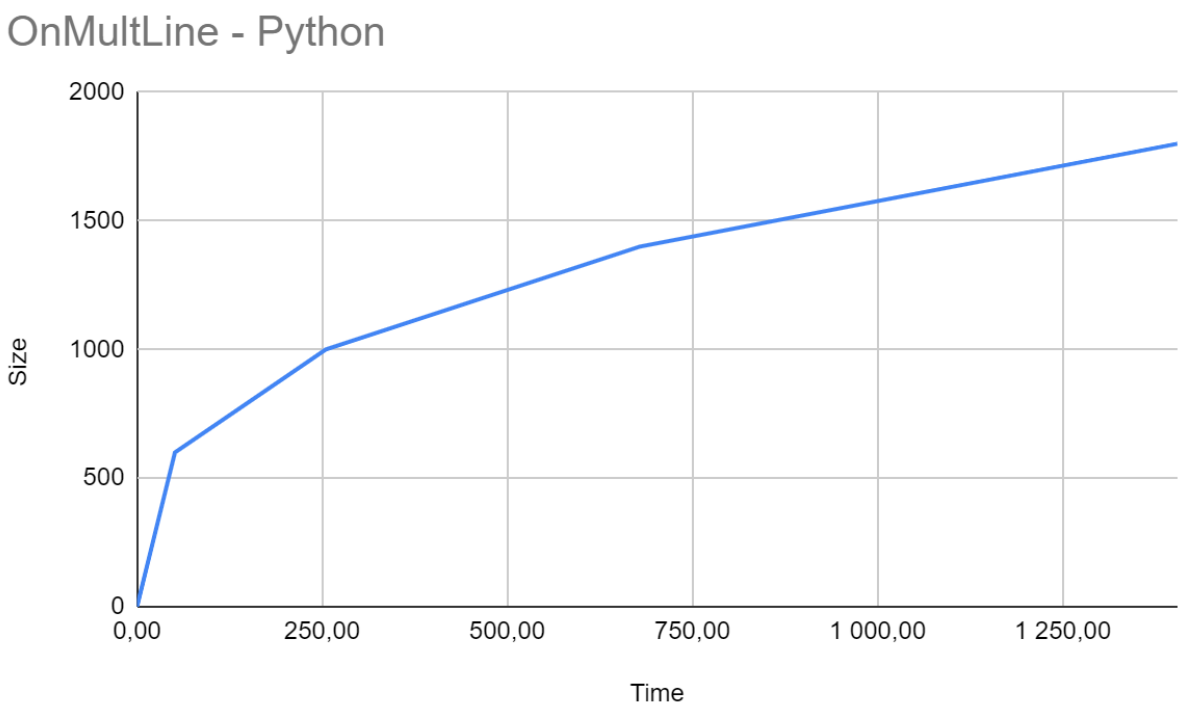


Algoritmo 2 - OnMultLine

C++

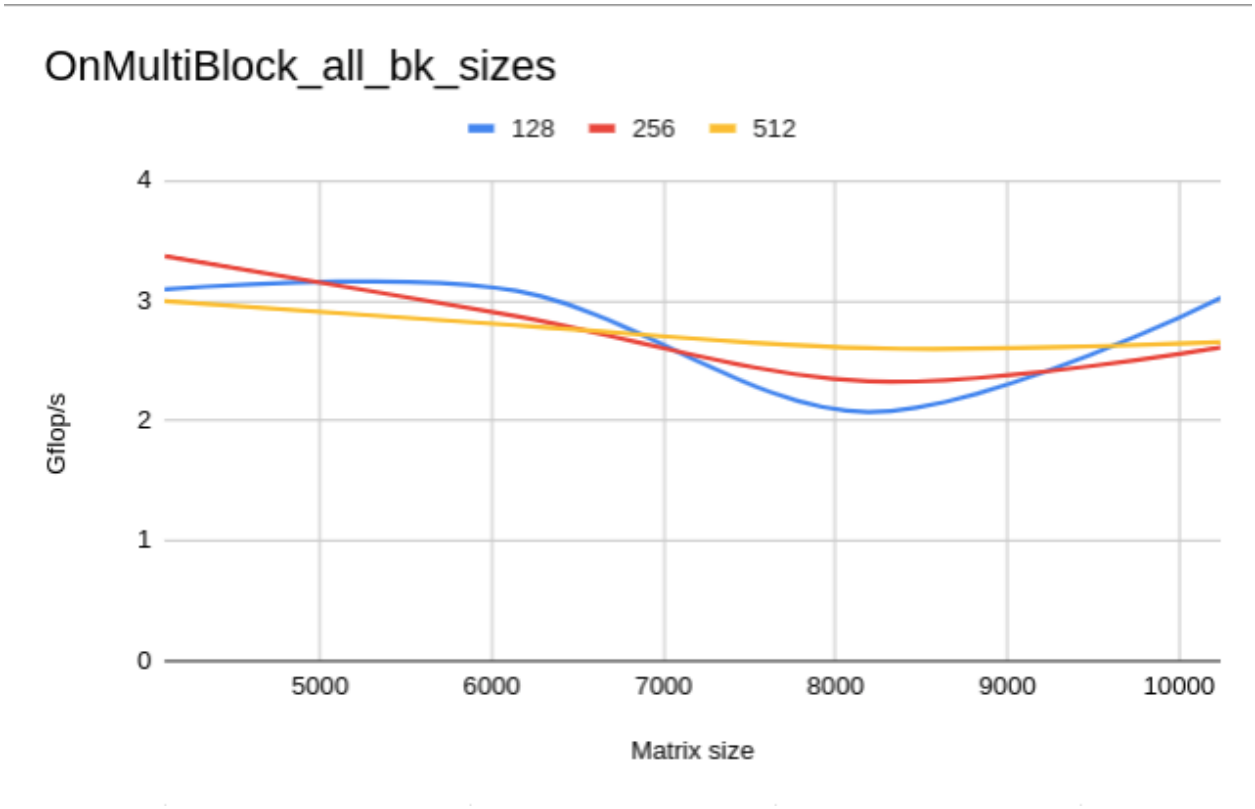


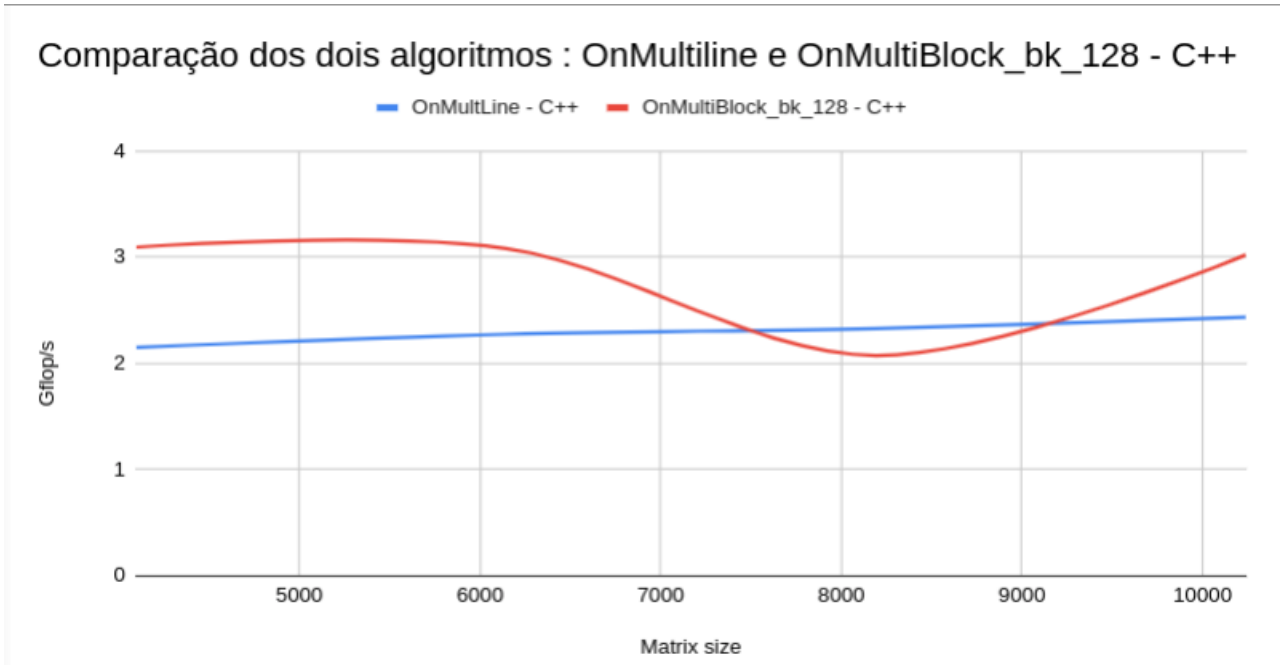
Python



Algoritmo 3 - OnMultBlock

C++


OnMultiBlock_all_bk_sizes
128
256
512
4
3
2
1
0
Gflop/s
5000
6000
7000
8000
9000
10000
Matrix size


Comparação dos dois algoritmos : OnMultiline e OnMultiBlock_bk_128 - C++
OnMultLine - C++
OnMultiBlock_bk_128 - C++
4
3
2
1
0
Gflop/s
5000
6000
7000
8000
9000
10000
Matrix size

4. Conclusão