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: OnMulti

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 $m \times p$.

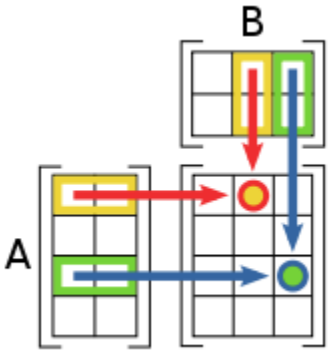
$$C = (AB)_{ij} = \sum_{r-1}^{n} a_{ir} b_{rj} = a_{i1} b_{1i} + a_{i2} b_{2i} + \ldots + a_{in} b_{nj}$$

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

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) : OnMultiLine

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:

		B	
		15	30
		8	4
A		C	
10	20	(10*15) + (20*8)	(10*30)+(20*4)
5	7		

1ª operação : (10*15) , 2ª operação: (10*30), 3ª operação : (20*8) e 4ª operação: (20*4)

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): OnMultBlock

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:

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

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

$$C = A * B = \left[\begin{array}{cc} C11 & C12 \\ C21 & C22 \end{array}\right] = \left[\begin{array}{cc} A11*B11+ & A11*B12+ \\ A12*B21 & A12*B22 \\ A21*B11+ & A21*B12+ \\ A22*B21 & A22*B22 \end{array}\right]$$

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];
                    }
                }
            }
        }
    }
}
```

Todos os algoritmos têm complexidade temporal O(n^3). Os dois primeiros facilmente é identificável já que são três ciclos for encadeados. No algoritmo por

blocos, apesar de termos 6 ciclos for, a forma como são iterados os três primeiros ciclos (bloco a bloco) e os últimos 3 ciclos mais interiores (também bloco a bloco), reduzem a complexidade também para O(n^3) . Haverão assim também 2*n^3 operações de floating point .

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 : “Level 1 data cache misses”
PAPI_L2_DCM : “Level 2 data cache misses ”
PAPI_L1_ICM : “Level 1 instruction cache misses ”
PAPI_L2_ICM : “Level 2 instruction cache misses”
PAPI_L1_TCM : “Level 1 cache misses ”
PAPI_L2_TCM : “Level 2 cache misses”
PAPI_TOT_INS : “Instructions completed”
```

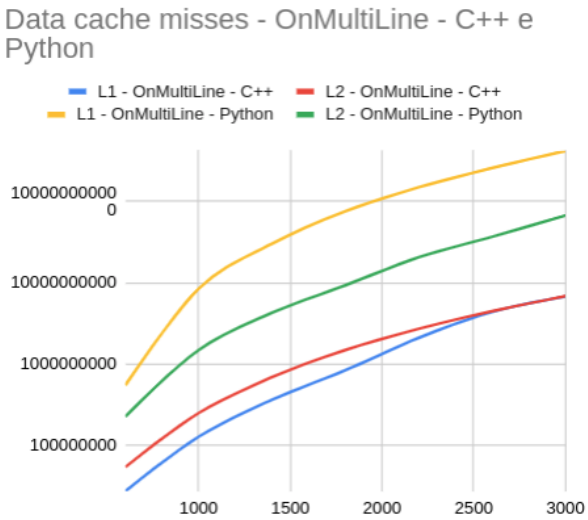
Gflop/s : ((2*n^3)/tempo_execução)/(10^9)

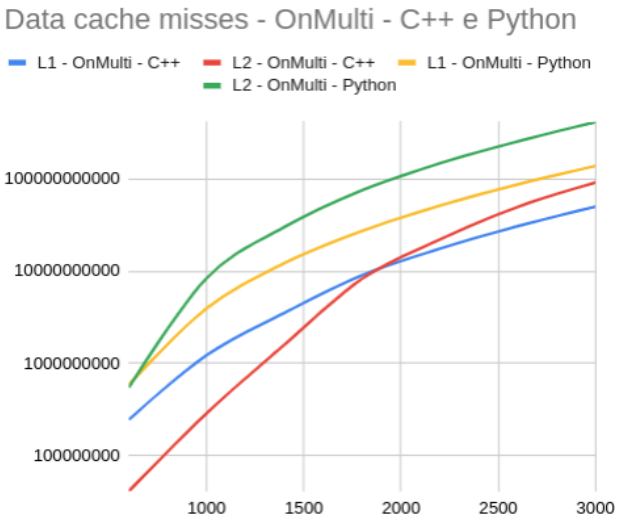
Utilizaremos as métricas data misses para L1 e L2 e Gflops/s , já que são os dados mais relevantes para avaliar as otimizações que utilizam a maneira como os processadores, cache e troca de dados funcionam para melhorar os algoritmos .

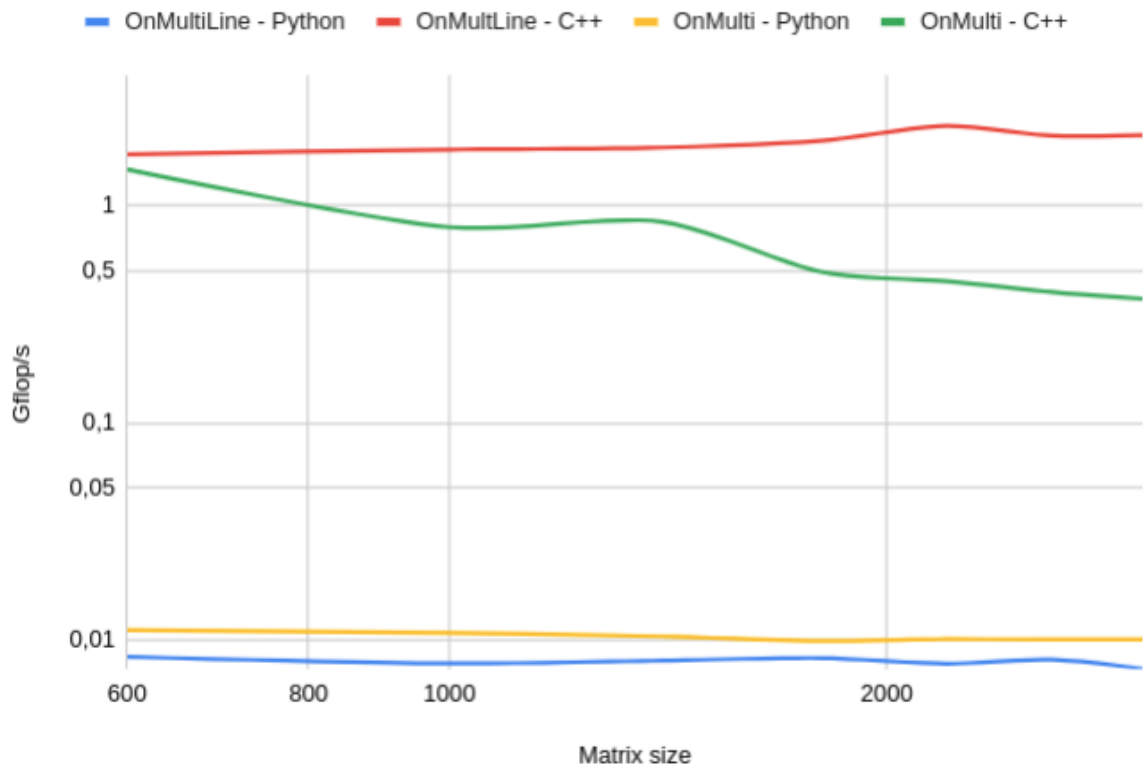
Os dados utilizados no trabalho são referentes a um processador Intel® Core™ i7-7600U com frequência base de 2.8GHz e máxima de 3.9GHz com 4MB de cache.

3. Resultados e Análise

Algoritmos - OnMulti e OnMulTiline
C++ e Python :







Foram utilizadas duas linguagens para os dois algoritmos : C++ e Python .

O algoritmo OnMultiLine ao contrário do OnMulti aproveita as características já em cima referidas : as localidades espacial e local da cache .

Localidade espacial, pois aproveita o facto de que estatisticamente quando um elemento é necessário existe uma probabilidade elevada de que os elementos próximos sejam também necessários nos próximos ciclos. Assim, vários elementos são carregados para a cache onde podem ser utilizados de uma forma mais célere .

Localidade temporal, já que os mesmos elementos sendo precisos, estatisticamente elementos que foram necessários no imediato, é bastante provável que sejam de novo requisitados.

Verificamos pelos gráficos que o algorimo OnMultiLine em C++ tem uma performance mais elevada (maior valor Gflop/s) que o OnMulti em C++ e ambos têm melhor performance que os seus pares em linguagem Python . A implementação do OnMultiline em C++, tem uma performance sustentada, visto que se mantêm relativamente estável à medida que a dimensão das matrizes aumenta . Ao contrário dos restantos algoritmos

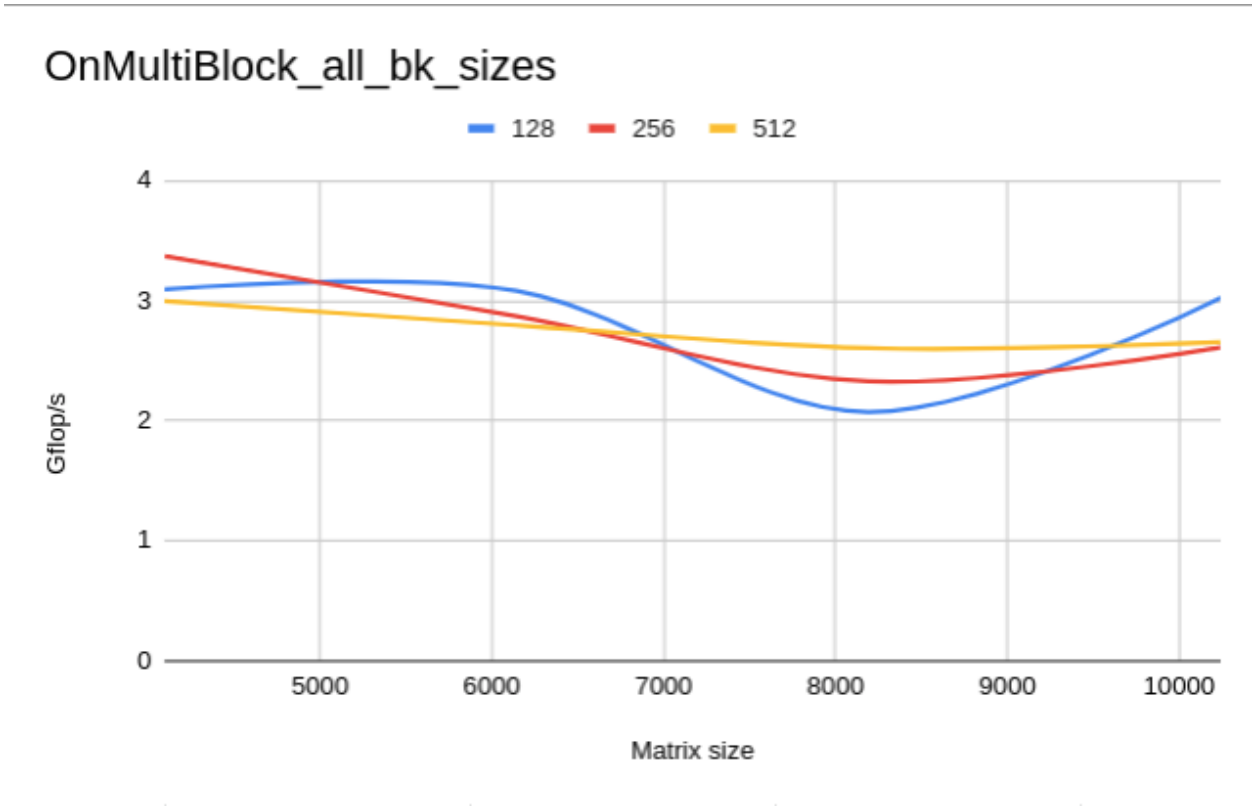
É também de salientar que o algoritmo OnMultLine em Python,ao contrário do que seria de esperar, não tem melhor performance que o OnMulti. Isto se deve ao

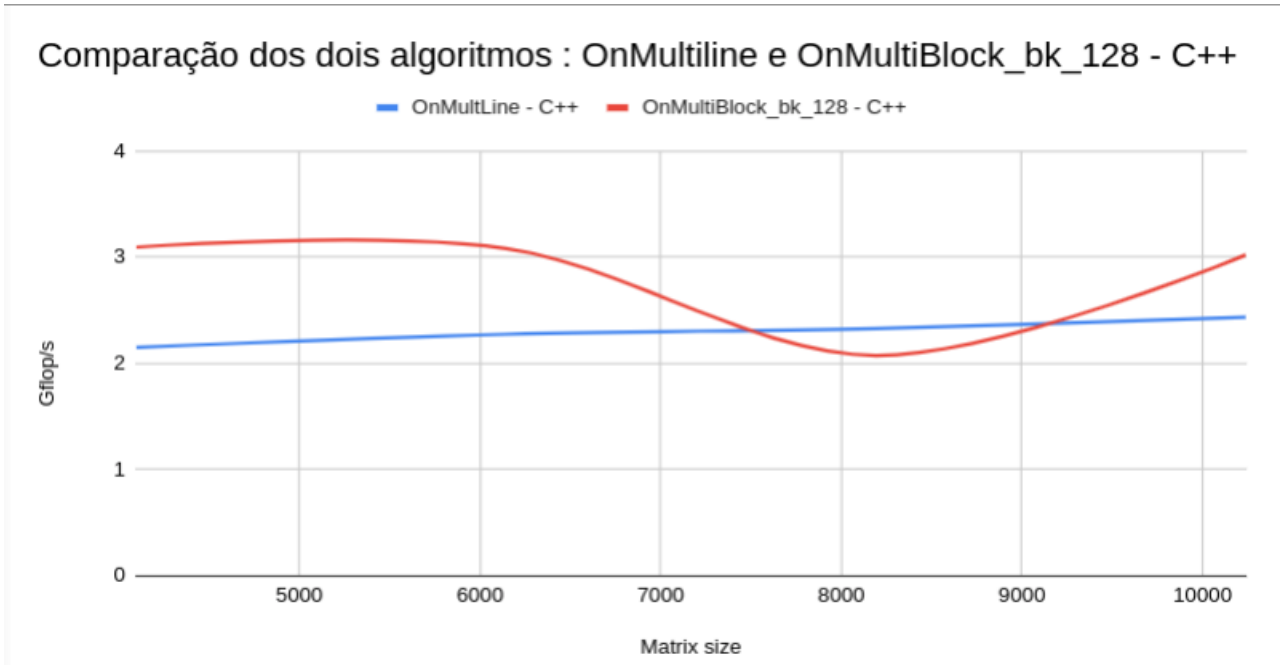
facto de em Python a alocação de memória em arrays não se passar em “row-major order como em C++. Não beneficiando assim das melhorias que a otimização permite para C++. Contudo, verificámos que a utilização de bibliotecas de Python como numpy podem ultrapassar esse problema .

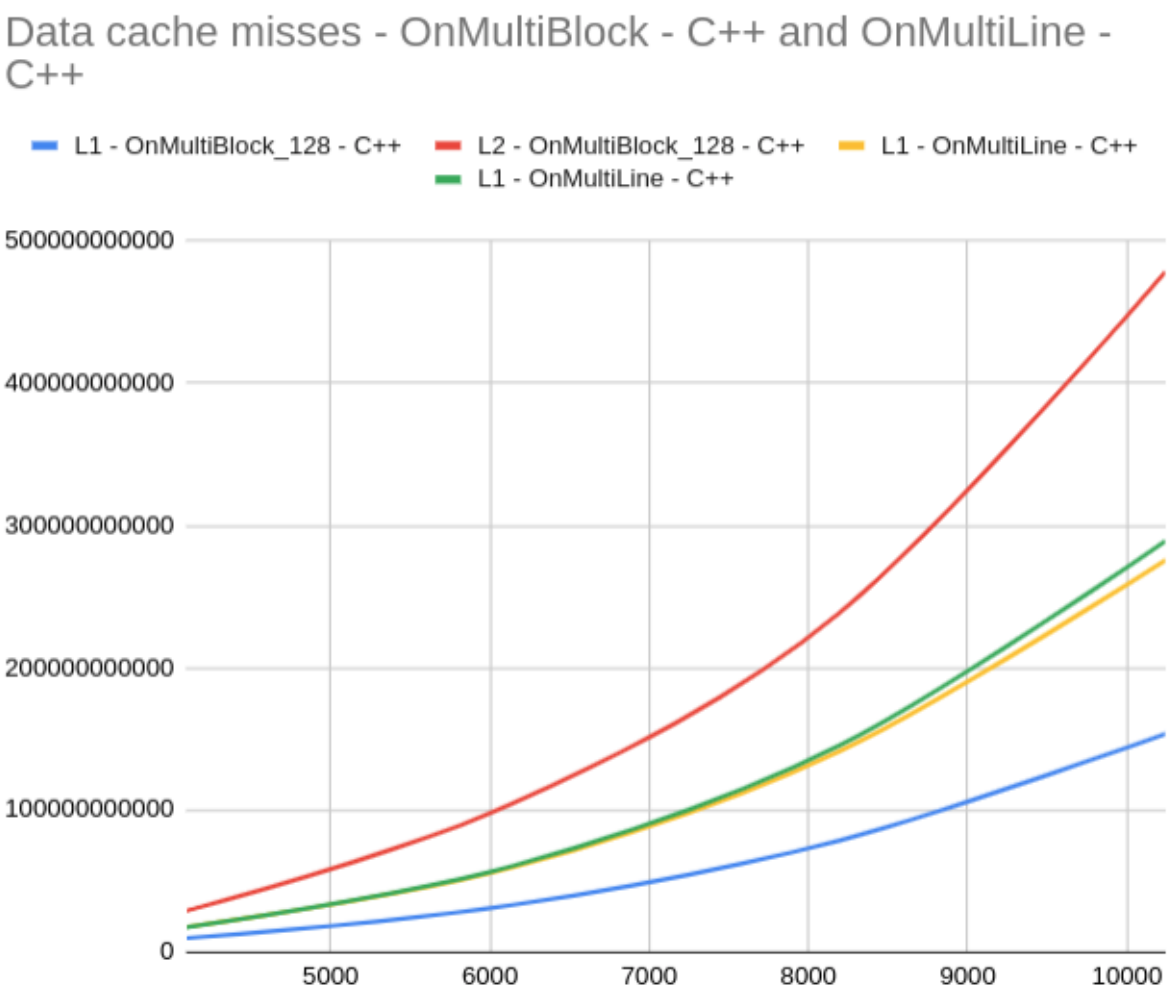
Em relação a data misses, como seria de esperar à medida que a dimensão das matrizes aumenta, vão aumentando exponencialmente, mas em menor escala na implementação dos algoritmos em C++, com valores ainda menores para o OnMultiLine . Apesar de haverem melhorias, isto tem um limite. Visto que mesmo que uma linha seja guarda na Cache, eventualmente se torna demasiado grande. O que leva a terem de haver cada vez mais acessos à memória princial . Voltando assim ao mesmo problema que o algoritmo OnMulti .

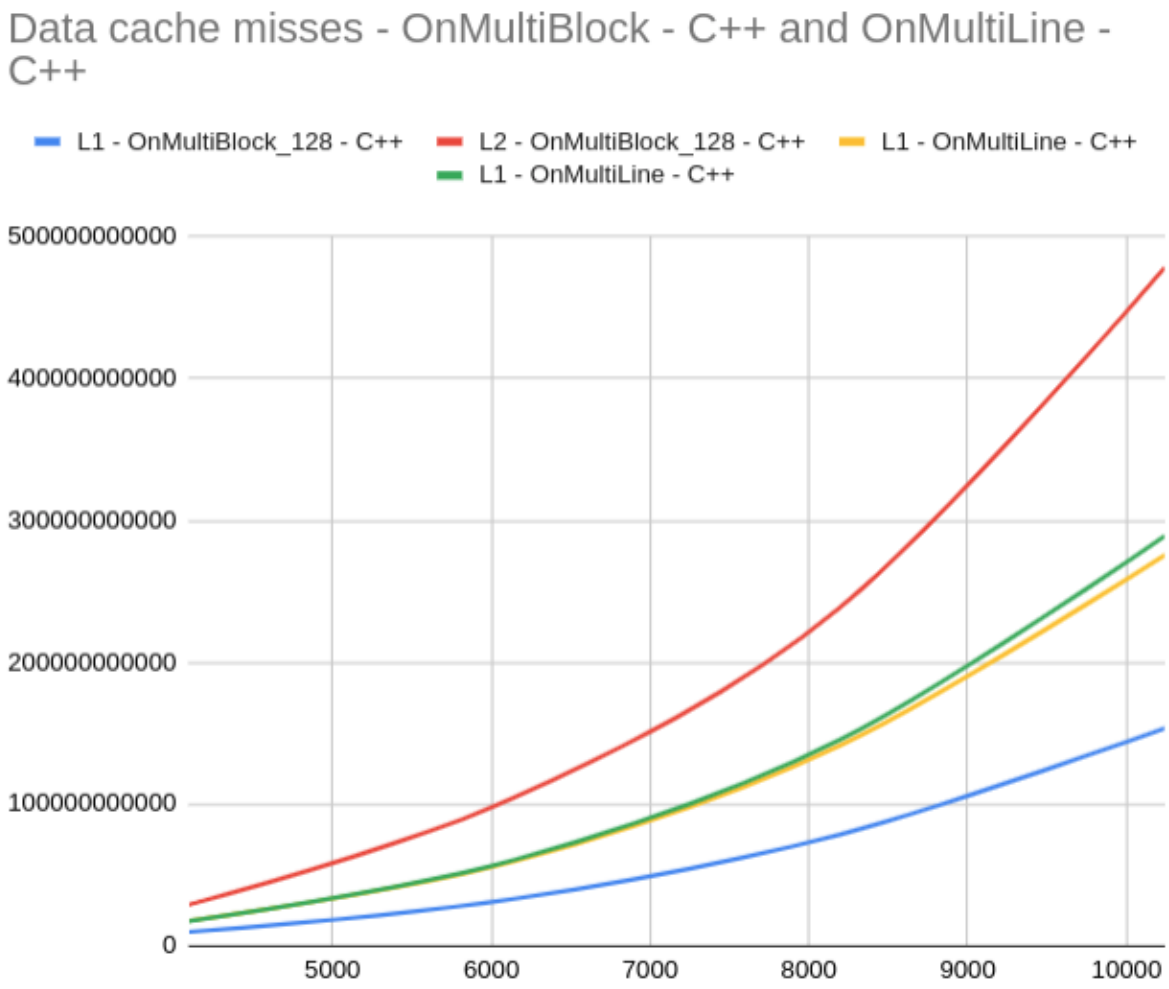
Verifica-se que em todos os casos, data misses L2 acaba sempre por ser superior ao quase igual ao data misses L1, o que é estranho, já que por norma tem uma maior Cache. Suspeitamos que mais algum tipo de evento seja contabilizado para L2 e que aumenta proporcionalmente com o tamanho da matriz . O mesmo comportamente acontece para o algoritmo OnMultiBlock .

Algoritmo 3 - OnMultBlock e OnMultiLine


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


Data cache misses - OnMultiBlock - C++ and OnMultiLine - C++
L1 - OnMultiBlock_128 - C++
L2 - OnMultiBlock_128 - C++
L1 - OnMultiLine - C++
L1 - OnMultiLine - C++
500000000000
400000000000
300000000000
200000000000
100000000000
0
5000
6000
7000
8000
9000
10000



Em relação ao algoritmo OnMultiBlock, primeiro comparámos para os tamanhos de blocos 128, 256 e 512. Verificando, que à medida que o tamanho das matrizes aumenta, o tamanho 128 é o melhor que se adequa o Processador e Cache utilizados neste trabalho .

Para tamanhos de blocos demasiado pequenos, não temos o comportamento esperado, já que não estamos a utilizar a capacidade toda que a Cache permite . Não assim obtendo a performance óptima que achar o tamanho de bloco óptimo para o processador utilizado permite .

Ao comparar com o algoritmo OnMultiLine em C++ verificamos que principalmente para tamanhos de matrizes muito grandes (acima de 9000) , o OnMultiBlock tem melhor performance, medida novamente em Gflops. Sendo a performance também sustentada, já que é estável, ficando ainda mais após o tamanho

de matrizes ser maior que 9000 (com aumento de performance), como se pode verificar no gráfico que compara com o algoritmo OnMultiLine .

Isto deve-se ao facto que a divisão das matrizes por blocos e ainda por cima otimizados para a Cache e processador utilizados, a linha que seria guardada na Cache e se tornaria demasiado grande, possivelmente após as matrizes terem dimensão maior de 9000 agora é dividida . O que permite até um certo ponto melhorar a performance em relação ao algoritmo OnMultiLine .

Tal como em cima foi descrito, aqui também as data cache misses aumentam exponencialmente à medida que o tamanho da matriz aumenta . Data misses L2 é maior que o L1 pelos mesmo motivos já explanados anteriormente .

O algoritmo MultiBlock ter mais data misses, com um aumento muito mais significativo para dimensões de matrizes acima de 9000, corrobora ainda mais o facto que deve ser aproximadamente nessa dimensão que a linha fica demasiado grande para ser guardada na Cache ,no algoritmo OnMultiLine, o que leva a ser vantajoso partir as matrizes por blocos, já que nesses blocos as linhas serão mais pequenas .

4. Conclusão

O mesmo código, pode ser optimizado de várias maneiras, de forma a utilizar da melhor forma os recursos e arquitetura de um computador moderno.

No caso da multiplicação de matrizes, modificar a forma como a mesma é feita, adaptando-a à forma como a Cache e o processador com as suas características funcionam e lidam com os dados é uma mais valia .

De facto a otimização de código permite ter ganhos significativos e no caso do algoritmo OnMultiBlock, para além de melhorar a performance é evidente que a partição das matrizes em blocos, se quisermos, permitirá depois por exemplo acrescentar programação paralela no cálculo dos blocos, o que pode melhorar ainda mais a performance .