FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

“Avaliação da performance de um core”

Mestrado Integrado em Engenharia Informática e Computação
Unidade Curricular de Computação Paralela
Ano Letivo 2015/2016

João António Custódio Soares 201206052
João Alberto Trigo de Bordalo Morais 201208217
30/03/2016

Índice

Descrição do Problema e Explicação dos Algoritmos

Na unidade curricular de Computação Paralela foi proposto estudar o efeito sobre o desempenho de um processador relativamente à hierarquia de memória quando se trata de grandes volumes de informação. Para fazer esta análise, iremos utilizar o produto entre duas matrizes quadradas.
No calculo do produto matricial usaremos dois algoritmos diferentes para efeitos de teste:

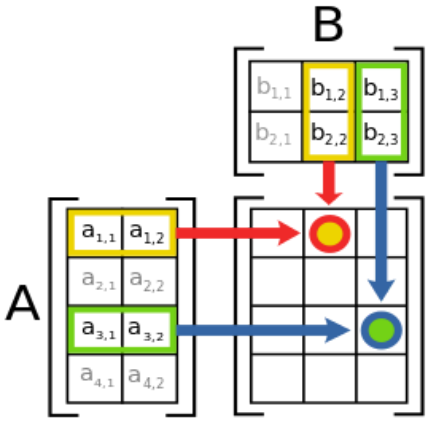
- **onMult:** este algoritmo calcula o produto entre duas matrizes da mesma dimensão da seguinte forma:

$$C_{ij} = (AB)_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} = a_{i0} b_{0j} + a_{i1} b_{1j} + \ldots + a_{i(n-1)} b_{(n-1)j}$$

Sendo:

- C_{ij} - Matriz resultado do produto matricial entre A e B;
- $(AB)_{ij}$ - Produto matricial entre as matrizes A e B;
- n - Tamanho das matrizes NxN;
- a_{ik} - Elemento da linha i da coluna k da matriz A;
- b_{kj} - Elemento da linha k da coluna j da matriz B.

Este algoritmo multiplica a primeira linha da matriz A por cada coluna da matriz B, como ilustra a seguinte imagem:



Legenda: Produto matricial entre a matriz A(4x2) e B(2x3).

O pseudocódigo para este algoritmo é o seguinte:

```
for(i=0; i<n; i++) {
        for( j=0; j<n; j++) {
                for( k=0; k<n; k++) {
                        C[i][j]= A[i][k] * B[k][j];
                }
        }
}
```

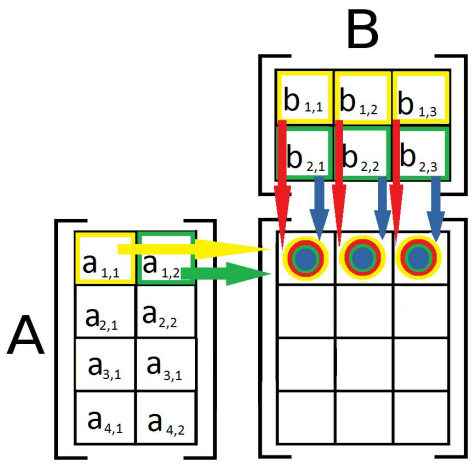
- **onMultLine:** este algoritmo calcula o produto matricial de forma diferente, relativamente ao anteriormente apresentado:

$$C_{ij} = (AB)_{ij} = \sum_{j=0}^{n-1} a_{ik} b_{kj} = a_{ik} b_{k0} + a_{ik} b_{k1} + \ldots + a_{ik} b_{k(n-1)}$$

Sendo:

- C_{ij} - Matriz resultado do produto matricial entre A e B;
- $(AB)_{ij}$ - Produto matricial entre as matrizes A e B;
- n - Tamanho das matrizes NxN;
- a_{ik} - Elemento da linha i da coluna k da matriz A;
- b_{kj} - Elemento da linha k da coluna j da matriz B.

Este algoritmo multiplica um elemento da matriz A pela linha correspondente da matriz B.



Legenda: Produto matricial entre a matriz A(4x2) e B(2x3).

O pseudocódigo para este algoritmo é o seguinte:

```
for(i=0; i<n; i++) {
        for( k=0; k<n; k++) {
                for( j=0; j<n; j++) {
                        C[i][j]= A[i][k] * B[k][j];
                }
        }
}
```

A grande diferença entre estes dois algoritmos, a nível de implementação, está na permutação entre o segundo e terceiro ciclo, o que, consequentemente, terá um impacto bastante significativo, algo a ser verificado na secção de Resultados e Análises.
Para este estudo três testes serão feitos recorrendo sempre a matrizes com o mesmo conteúdo:

- **Matriz A:** corresponde à matriz identidade;
- **Matriz B:** o conteúdo de cada elemento corresponde ao valor da respetiva linha +1.

Os testes:

1. O primeiro teste consiste em fazer o estudo dos tempos que o processador demora a calcular o produto entre duas matrizes quadradas com o algoritmo *onMult*. Para tal implementou-se o algoritmo em duas linguagens de programação: *c++* e *java*. As matrizes utilizadas para este teste são: 600x600, 1000x1000, 1400x1400, 1800x1800, 2200x2200, 2600x2600 e 3000x3000, sete matrizes diferentes.
2. O segundo teste segue o mesmo princípio que o anterior, porém, utiliza o algoritmo *onMultLine*. Adicionalmente, também faz o calculo matricial para as matrizes: 4000x4000, 6000x6000, 8000x8000, 10000x10000 para verificar o comportamento deste algoritmo perante um nível muito mais elevado de dados.
3. O último teste pretende recorrer à utilização do *OpenMP* para implementar versões paralelas dos dois algoritmos e dividir as operações aritméticas entre 1 a 4 *threads*, registando o tempo de execução das matrizes de dimensão compreendida entre 600x600 e 3000x3000 com 400 unidades de incremento entre iterações.

Métricas de Desempenho e Metodologias de Avaliação

As métricas de desempenho usadas nos testes foram as seguintes:

- **Tempo de execução** em função da **dimensão das matrizes**;
- **Número *Cache Misses*** em função da dimensão das matrizes, através dos contadores da ferramenta *PAPI* **L1_DCM** e **L2_DCM**;
- **Número de instruções em vírgula flutuante (*GFLOPS*)** em função da **dimensão das matrizes**;
- **Número de *threads*** utilizadas para as três primeiras métricas em função da **dimensão das matrizes**.

A metodologia de avaliação utilizada foi a de **comparação**. A referência de comparação é estabelecida pelo algoritmo *onMult* na linguagem *c++* comparando com o mesmo algoritmo em *java*; comparando com o algoritmo *onMultLine* em *c++*; comparando com o mesmo algoritmo em *java*; comparando o algoritmo *onMult* na linguagem *c++* usando múltiplas *threads*; e comparando o algoritmo *onMultLine* na mesma linguagem e usando múltiplas *threads*.
O propósito desta metodologia de avaliação é compreender o impacto que pequenas alterações na execução de código têm a nível da utilização memória *cache* e consequentes variações no tempo de execução e performance.

Características do computador de testes:
Modelo Processador: Intel(R) Core (™) i7-4790 CPU @ 3.60GHz
Nº Processadores: 8
Tamanho Cache: 8192 KB
Memória Ram total: 16342672 KB

Resultados e Análises

Para o primeiro teste, fez-se a comparação do tempo de execução em função da dimensão da matriz para as linguagens de programação *c++* e *java* utilizando o algoritmo *onMult.*
Analisando o gráfico 1 é possível retirar duas conclusões: com o aumento da dimensão da matriz, o tempo de execução aumenta de forma exponencial; e ainda, também com o aumento da dimensão da matriz, o algoritmo em *java* tem um tempo de execução menor que o mesmo algoritmo em *c++*, consequentemente, melhor performance.
No gráfico 2 retira-se que o número de cache *misses* tem um comportamento semelhante ao tempo de execução: com o aumento da dimensão da matriz, o número de *cache misses* aumenta de forma exponencial, em contrapartida, os *GFLOPS* diminuem com o aumento da dimensão da matriz.

Análise do primeiro teste:

Confrontando os dois gráficos, podemos concluir que o algoritmo em *java* é mais eficaz para tratar este volume de dados e que o algoritmo em *c++* tem uma diminuição de performance mais acentuada à medida que o espaço do problema aumenta. Com o aumento do tamanho das linhas das matrizes, maior a capacidade de memória necessária para as armazenar em *cache*, o que implica um menor número de linhas armazenadas em *cache* em cada instante. Como este algoritmo não toma partido da utilização da memória *cache* (em que a informação é carregada linha a linha), ao iterar pelas colunas da matriz B, reduz a probabilidade de reutilização de cada bloco de dados carregado, aumentando exponencialmente o número de *cache misses*, sendo assim necessário mais instruções não aritméticas durante a execução, o que é confirmado pelo declínio nos *GFLOPS* e deprecação do desempenho.

Para o segundo teste, em relação ao primeiro, difere no algoritmo. No gráfico 3 e 5 constata-se que, com o aumento da dimensão da matriz, os tempos de execução são menores comparativamente ao primeiro teste, e ainda que o *c++* tem uma performance melhor no cálculo do produto matricial.
Nos gráficos 4 e 6 verifica-se que, relativamente ao primeiro teste, o número de *cache misses* é menor e que o número de *GFLOPS* é superior, mesmo assim, o comportamento é o mesmo: o número de *cache misses* aumenta de forma exponencial e os *GFLOPS* diminui com a dimensão da matriz.

Análise do segundo teste:

Confrontando os dois testes, em concorrência com as duas linguagens de programação, salienta-se que o algoritmo utilizado tem um enorme impacto na performance. Isto indica que o algoritmo *onMultLine* faz um aproveitamento muito superior do conteúdo da cache - menor número de cache *misses* e, consequentemente, maior valor de GFLOPS. Com o aumento do volume de dados, o desempenho do algoritmo em *c++* torna-se superior ao do algoritmo em *java*. Isto dá-se pois o *c++* é uma linguagem de mais baixo nível logo é mais eficaz na execução de cada instrução, que se torna um fator crítico para o desempenho devido ao contínuo aumento exponencial de *cache misses*.

O terceiro teste adiciona a métrica número de *threads* para avaliar o impacto do uso de paralelismo na performance do cálculo do produto matricial. Usando o algoritmo *onMult* e até quatro *threads*, o tempo de execução vai diminuindo com o maior número de *threads* e, adicionalmente, com o aumento da dimensão da matriz, a curva é menos acentuada, como é possível verificar no gráfico 7. Como esperado pela análise dos dois primeiros testes, o algoritmo *onMultLine* tem um tempo de execução menor, relativo ao algoritmo *onMult*, e com o uso de *threads* tem um desempenho melhor. Contudo, este desempenho não melhora com o aumento do número de threads. Os outros gráficos, 8, 9,11 e 12, têm um comportamento semelhante aos dos testes 1 e 2 com variações nos seus valores devido ao facto de estar a ser usado paralelismo.

Análise do terceiro teste:

O aumento do número de *threads* é um fator predominante na performance do cálculo do produto matricial, naturalmente, pois o “trabalho” é dividido pelas várias *threads*. Contudo, comparando entre os dois algoritmos, para o algoritmo *onMultLine*, quando mais de duas *threads* são usadas há um declínio de performance. Este declínio deve-se ao facto de haver *overhead* e “atropelamento” das *threads* ao acederem *às* informações na cache. Uma vez que este algoritmo, por aproveitar melhor o conteúdo carregado para cache, fá-lo ser mais eficiente até certo ponto. No entanto, preve-se que para uma dimensão da matriz bastante superior à dos testes, o aumento do número de threads melhore o desempenho do cálculo.

Conclusões

Após uma análise aos resultados dos três testes podemos afirmar que o algoritmo *onMultLine* tira melhor partido da *cache,* tendo menos *cache misses* e, no geral, melhor desempenho. Também podemos observar que a partir de certo ponto, o número de *GFLOPS* mantém-se aproximadamente constante. Isto deve-se ao número de instruções ser diretamente proporcional ao tempo de execução. A explicação para este facto é que o número de instruções em vírgula flutuante (operações aritméticas) e o tempo gasto pelo processador nelas terão um impacto constante a partir de uma grande dimensão da matriz, pois o processador não conseguirá calcular mais rápido as mesmas. Finalmente, podemos também afirmar que o *java*, mais concretamente, a *JVM* (*Java Virtual Machine)* gere de forma mais eficaz as alocações de memória, em comparação ao *c++,* mas que é menos eficaz a executar as instruções. Isto justifica-se com o tempo de execução do algoritmo *onMult* ser menor em *java*, mas maior quando é feita uma boa gestão da *cache,* como nos mostra a experiência 2.

Concluímos então que a boa utilização da *cache* é crucial para obter máximo desempenho ao diminuir o tempo gasto em operações de *read/write*, operações estas que não são possíveis de optimizar recorrendo a técnicas de paralelismo. Concluímos também que as técnicas de paralelismo introduzem uma nova complexidade ao problema devido ao *overhead* introduzido pela sincronização das *threads,* logo a sua utilização deve ser adaptada à dimensão dos dados de modo a obter máximo desempenho.

Com a realização deste estudo conseguimos entender a influência de uma utilização optimizada da memória *cache* e de paralelismo em problemas de complexidade exponencial. Também foi proveitoso para obter um melhor entendimento do funcionamento da memória *cache* e para perceber que tipo de características devemos procurar nas linguagens de programação para que a escolha seja mais adequada ao problema em questão.

Infelizmente, gostariamos de ter acesso ao contador do PAPI_L1_DCA e PAPI_L2_DCA (data cache accesses) para determinar a percentagem de cache misses em função do cache *acesses* para melhor avaliar a necessidade de recorrer às seguintes caches e, posteriormente, memória Ram. Tal acesso foi impossível pois o computador usado para os testes não possuia o contador PAPI_L1_DCA, nem o PAPI_L1_DCH (data cache hits). Mesmo assim, com os dados retirados, foi possível tirar as mesmas conclusões.

Anexo

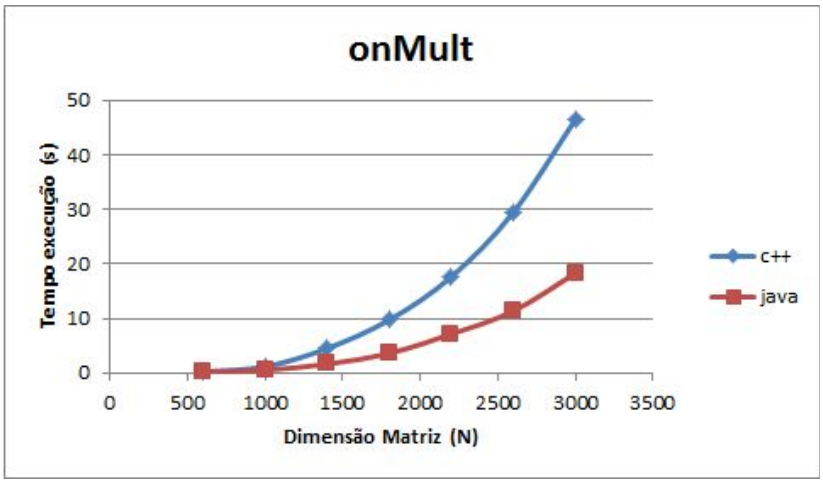


Gráfico 1: gráfico de dispersão representativo dos tempos de execução em função da dimensão da matriz do algoritmo *onMult* em *c++* e *java*

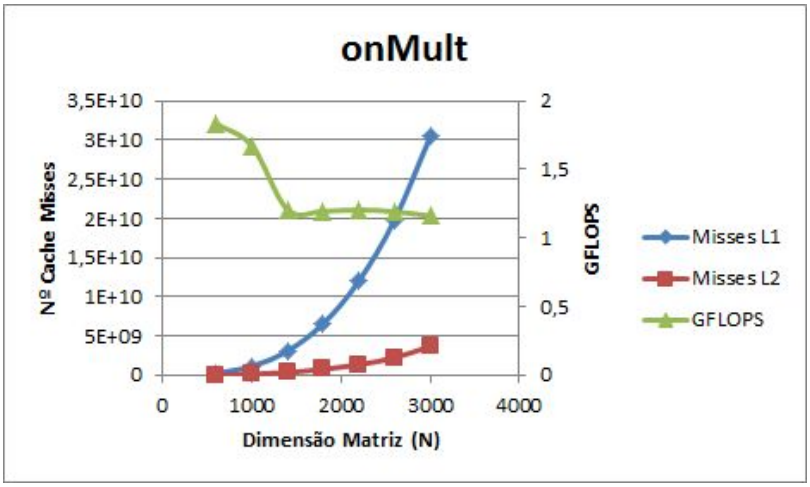


Gráfico 2: gráfico de dispersão representativo do número de cache *misses* e GFLOPS em função da dimensão da matriz do algoritmo *onMult* em *c++*

	c++	L1	L2		*java*
N	Algoritmo a	*Cache Misses*	*Cache Misses*	*GFLOPS*	Algoritmo a
600	0.235876	244786966	27390575	1.831470773	0.176612106
1000	1.19737	1129277006	126275914	1.670327468	0.535022524
1400	4.54923	3095414206	344402300	1.206357999	1.652178613
1800	9.75962	6580589845	735976891	1.195128499	3.637535775
2200	17.6603	12000741930	1341747512	1.20586853	7.156426099

2600	29.4776	19803559616	2228998572	1.192498711	11.37579704
3000	46.48	30415731832	3664437564	1.161790017	18.30811174

Tabela 1: levantamento de dados referentes aos gráficos 1 e 2.

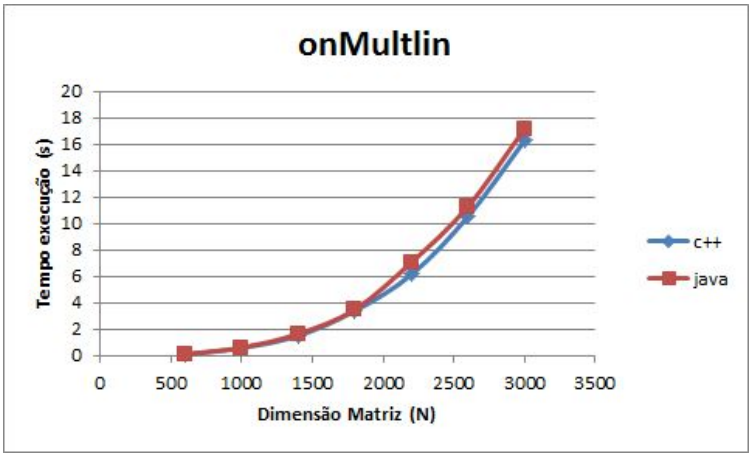


Gráfico 3: gráfico de dispersão representativo dos tempos de execução em função da dimensão da matriz do algoritmo *onMultlin* em *c++* e *java*

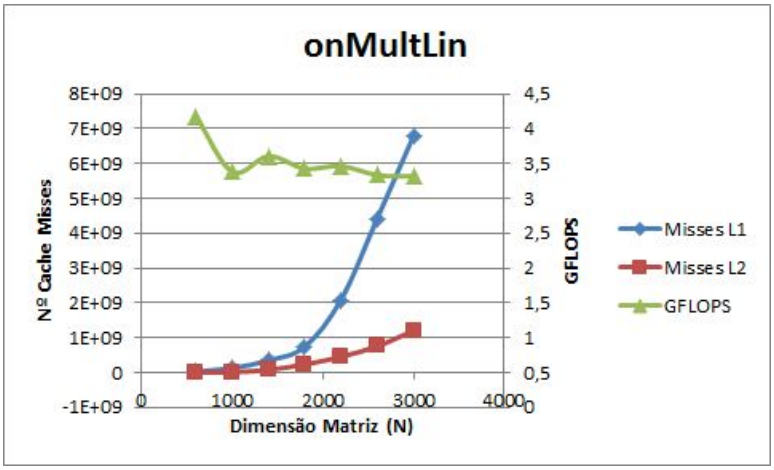


Gráfico 4: gráfico de dispersão representativo do número de cache *misses* e GFLOPS em função da dimensão da matriz do algoritmo *onMultlin* em *c++*

	c++	*L1*	*L2*		*java*
N	*Algoritmo b*	*Cache Misses*	*Cache Misses*	*GFLOPS*	*Algoritmo b*
600	*0.10345*	*27149280*	*807552*	*4.175930401*	*0.130240998*
1000	*0.591527*	*125645371*	*6508827*	*3.381079815*	*0.618105435*
1400	*1.5222*	*345121305*	*75578195*	*3.605308107*	*1.666501341*
1800	*3.40288*	*738024427*	*231113623*	*3.427684785*	*3.508080999*
2200	*6.14865*	*2082745068*	*452130600*	*3.463524514*	*7.073708764*
2600	*10.5439*	*4421143189*	*752701010*	*3.333870769*	*11.29439365*
3000	*16.2839*	*6791327626*	*1200191807*	*3.316158905*	*17.17380558*

Tabela 2: levantamento de dados referentes aos gráficos 3 e 4.

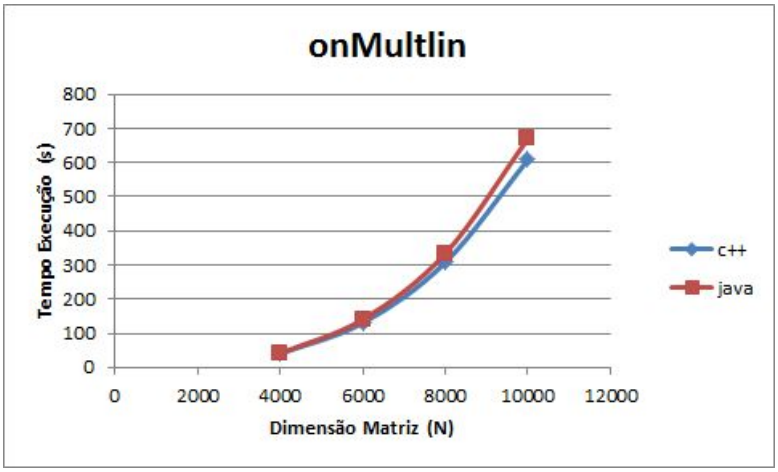


Gráfico 5: gráfico de dispersão representativo dos tempos de execução em função da dimensão da matriz do algoritmo *onMultlin* em *c++* e *java.* Neste caso as dimensões das matrizes são superiores de modo a poder estudar o comportamento do algoritmo quando se trata de grandes volumes de dados.

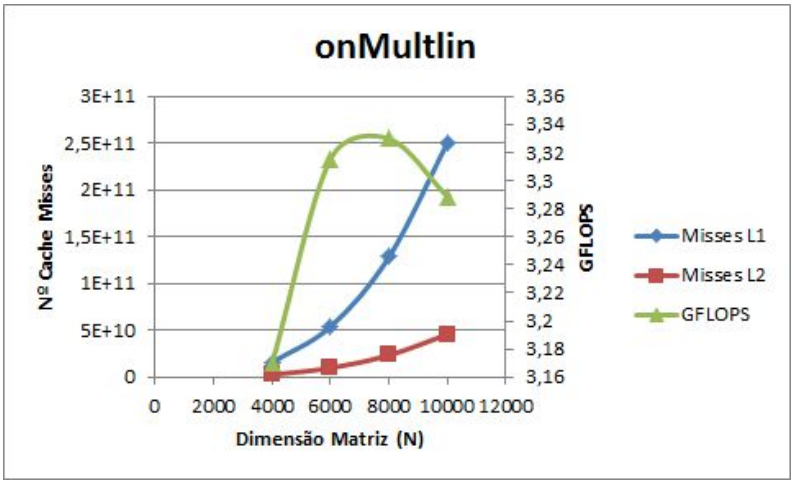


Gráfico 6: gráfico de dispersão representativo do número de cache *misses* e GFLOPS em função da dimensão da matriz do algoritmo *onMultlin* em *c++.* Neste caso as dimensões das matrizes são superiores de modo a poder estudar o comportamento do algoritmo quando se trata de grandes volumes de dados.

	c++	L1	L2		*java*
N	Algoritmo b	*Cache Misses*	*Cache Misses*	*GFLOPS*	Algoritmo b
4000	40.3765	16091193741	2861024567	3.170160861	41.6045144
6000	130.298	54199687221	9933034141	3.31547683	140.8815447
8000	307.525	128607100409	23572880589	3.329810585	333.5880108
10000	608.192	250625403247	45969194469	3.288435231	670.1669272

Tabela 3: levantamento de dados referentes aos gráficos 5 e 6.

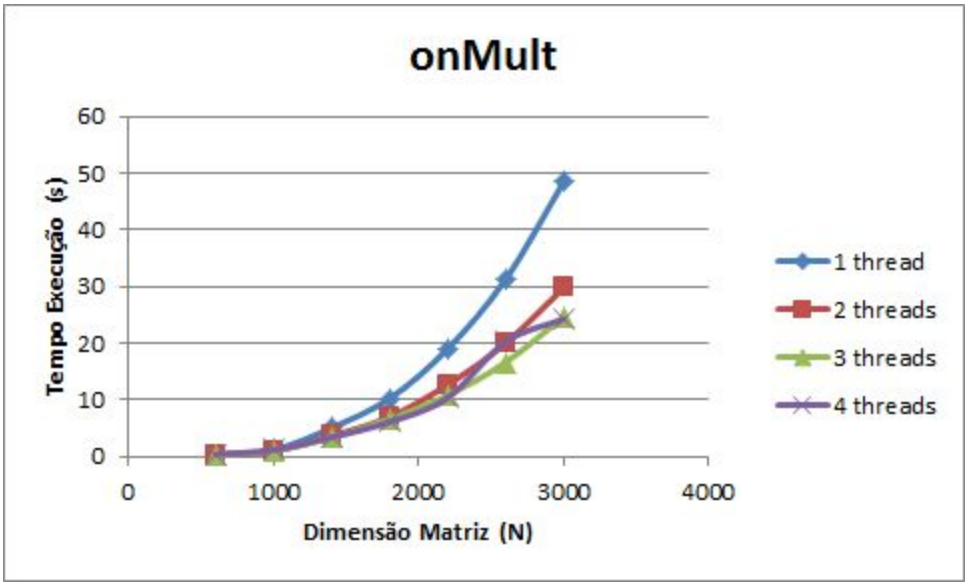


Gráfico 7: gráfico de dispersão representativo dos tempos de execução em função da dimensão da matriz do algoritmo *onMult* em *c++* recorrendo à utilização de uma, duas, três e quatro *threads*.

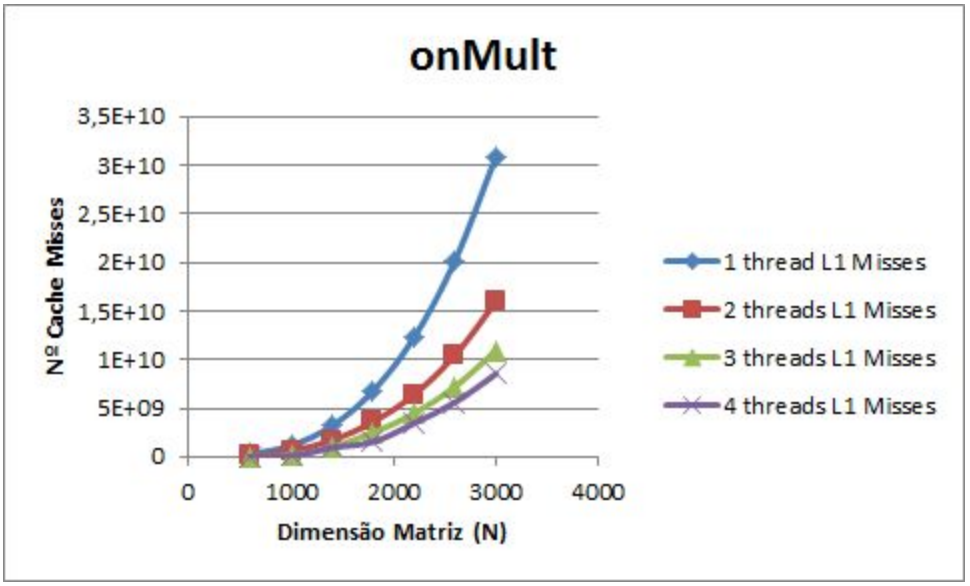


Gráfico 8: gráfico de dispersão representativo do número de cache *misses* em função da dimensão da matriz do algoritmo *onMult* em *c++* recorrendo à utilização de uma, duas, três e quatro *threads*.

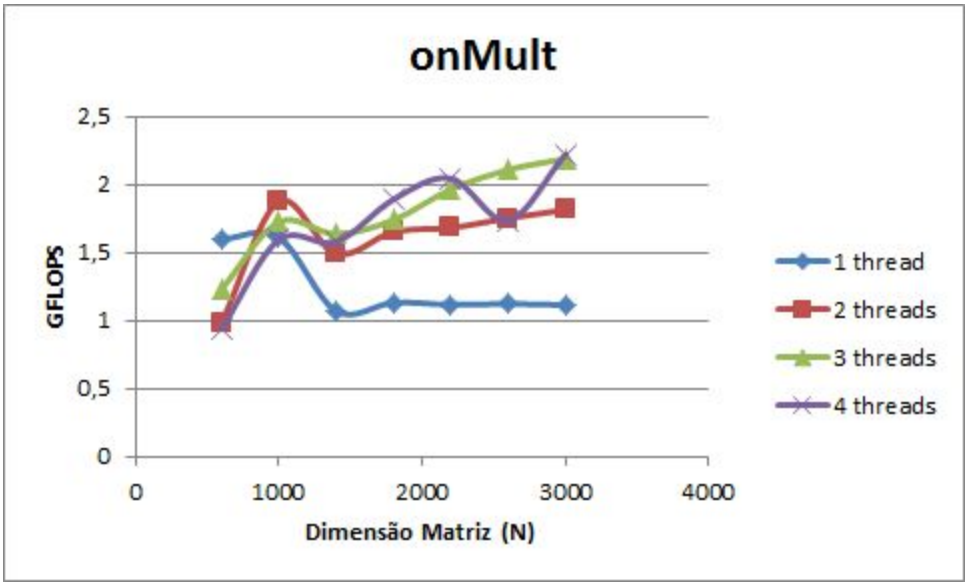


Gráfico 9: gráfico de dispersão representativo do número de GFLOPS em função da dimensão da matriz do algoritmo *onMult* em *c++* recorrendo à utilização de uma, duas, três e quatro *threads*.

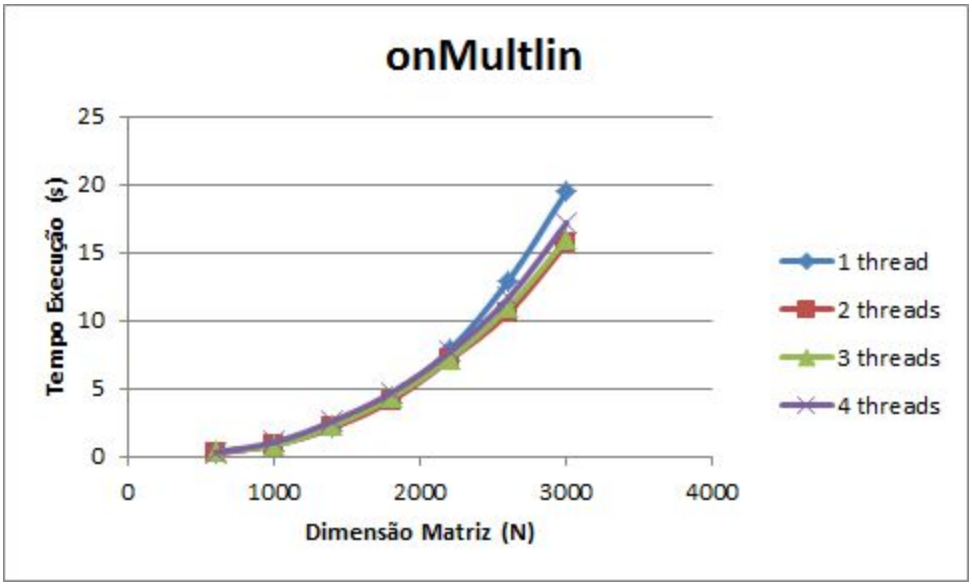


Gráfico 10: gráfico de dispersão representativo dos tempos de execução em função da dimensão da matriz do algoritmo *onMultlin* em *c++* recorrendo à utilização de uma, duas, três e quatro *threads*.

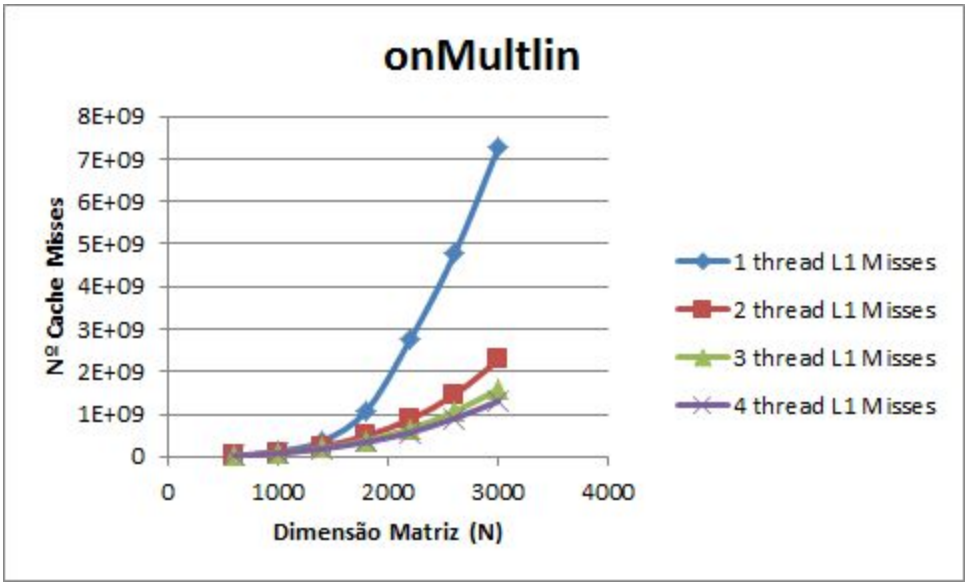


Gráfico 11: gráfico de dispersão representativo do número de cache *misses* em função da dimensão da matriz do algoritmo *onMultlin* em *c++* recorrendo à utilização de uma, duas, três e quatro *threads*.

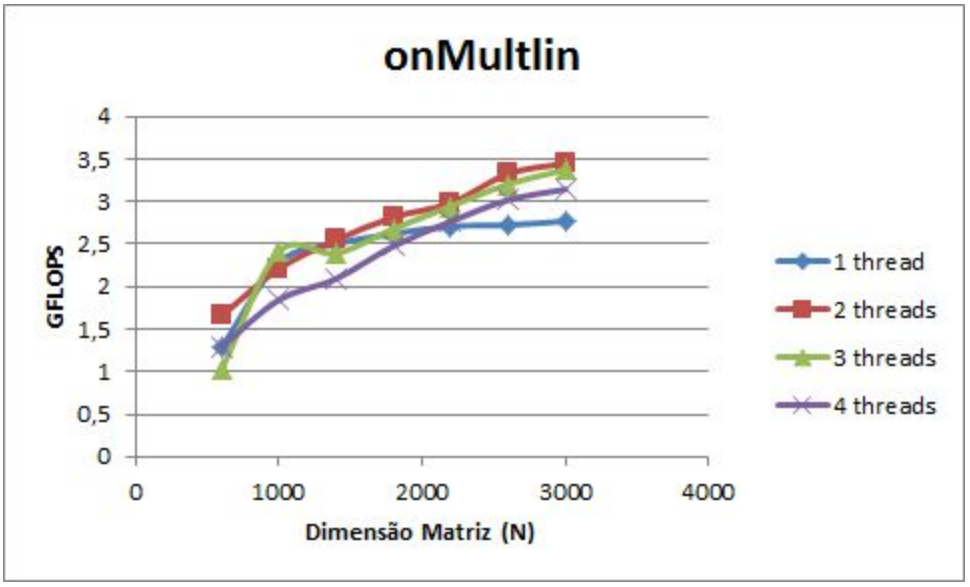


Gráfico 12: gráfico de dispersão representativo do número de *GFLOPS* em função da dimensão da matriz do algoritmo *onMultlin* em *c++* recorrendo à utilização de uma, duas, três e quatro *threads*.

		c++	L1	L2		c++	L1	L2	
th	N	Algoritmo a	Misses	Misses	GFLOPS	Algoritmo b	Misses	Misses	GFLOPS
1	600	0.27022	275255788	34047884	1.598697358	0.337153	27318475	631506	1.281317384
	1000	1.23957	1180034569	125415416	1.613462733	0.877123	129322321	4955253	2.280181913
	1400	5.12406	3197930189	345608587	1.07102571	2.19435	368532517	56617787	2.500968396
	1800	10.3099	6795765474	774207753	1.13133978	4.44661	1094399048	198642686	2.623121884
	2200	19.0315	12264148694	1355572798	1.118986943	7.86326	2778172809	403056958	2.708291472
	2600	31.2838	20168645109	2272137705	1.123648662	12.9035	4773216392	731713966	2.72422211
	3000	48.4399	30845455724	4373405049	1.114783474	19.494	7285487824	1083609674	2.770083102
2	600	0.440122	108409838	27253511	0.9815460259	0.260918	25259391	10835717	1.655692593
	1000	1.0678	607103437	80631262	1.873009927	0.905072	98024219	30557528	2.209768947
	1400	3.68205	1709208567	207779874	1.490474057	2.15448	248682061	98516454	2.547250381
	1800	7.0589	3593486295	410799063	1.652382099	4.1404	496546525	215395033	2.817119119
	2200	12.6602	6401423853	757482595	1.682121925	7.14181	887069561	388852175	2.981877143
	2600	20.0583	10457529619	1225820105	1.752491487	10.5716	1464019581	638683119	3.325135268
	3000	29.7826	15925382565	1861436716	1.813139216	15.6555	2283452085	927662708	3.449267031
3	600	0.351067	45141167	21548161	1.230534342	0.423346	17095363	7122725	1.020441908
	1000	1.16028	149488038	66560634	1.72372186	0.829315	85346855	33986857	2.411628874
	1400	3.353	1019196770	160124249	1.636743215	2.29558	203462867	103183869	2.390681222
	1800	6.70654	2498272195	318157825	1.739197858	4.3526	388721206	201151761	2.679777604
	2200	10.8416	4450038827	563214097	1.964285714	7.23359	662080070	350013704	2.944042999
	2600	16.6742	7218455771	890586699	2.108167108	10.9744	1067282377	549928865	3.20309083
	3000	24.6776	10935103352	1337835768	2.188219276	16.0106	1591826569	812629589	3.372765543
4	600	0.460132	31670189	17668221	0.9388610225	0.335074	20542240	10056418	1.289267445
	1000	1.25479	97845949	61578549	1.593892205	1.08565	81137240	37644546	1.842214342

	1400	3.47779	944889698	122708699	1.578013624	2.62344	190962169	106016920	2.090909859
	1800	6.16823	1537926562	275808326	1.89098007	4.71148	348687415	187443282	2.475655208
	2200	10.4102	3448819937	483843913	2.045685962	7.70077	572948772	303008642	2.765437742
	2600	20.2701	5604652604	719134182	1.7341799	11.6511	908004492	476204957	3.017054184
	3000	24.3232	8532420444	1116126768	2.220102618	17.1831	1311368633	680454035	3.142622693

Tabela 4: levantamento de dados referentes aos gráficos 7, 8, 9, 10, 11 e 12.