Faculdade de Engenharia da Universidade do Porto

Relatório do 1º Trabalho Laboratorial

Avaliação da performance de um *core*

Computação Paralela e Distribuída, Março de 2022 - L.EIC:

Professor Jorge Manuel Gomes Barbosa

Professor Pedro Alexandre Guimarães Lobo Ferreira Souto

Professor Pedro Miranda de Andrade de Albuquerque d'Orey

Grupo 3LEIC03G04 (Estudantes & Autores**):**

Henrique Ribeiro Nunes up201906852@up.pt

Patrícia do Carmo Nunes Oliveira up201905427@up.pt

Rafael Fernando Ribeiro Camelo up201907729@up.pt

Sumário

Este relatório incide sobre o primeiro trabalho laboratorial realizado. Tal trabalho tem como propósito principal a implementação de 3 algoritmos diferentes para o cálculo de multiplicação de matrizes tomando partido de apenas um *core* e a avaliação da sua performance.

O relatório pretende, não só apresentar e analisar os algoritmos implementados, mas também detalhar o tratamento dos dados e medidas de comparação utilizadas e as conclusões obtidas.

Introdução

O trabalho foi realizado no âmbito da unidade curricular de Computação Paralela e Distribuída, do ano letivo de 2021/22, com o objetivo de implementar, compreender e avaliar a performance de um *core* para diferentes versões de algoritmos para a resolução do mesmo problema.

O relatório tem como objetivo explorar as estratégias utilizadas no desenvolvimento do trabalho. Nas seguintes secções encontram-se as seguintes informações:

- *Descrição do Problema*: apresentação do caso de estudo;
- *Descrição dos Algoritmos Implementados*: exploração das versões do algoritmo;
- *Medidores de Performance*: identificação e justificação dos parâmetros de performance selecionados;
- *Resultados Obtidos e Análise*: apresentação e análise dos resultados obtidos - avaliação e comparação das versões.

Descrição do problema

O **problema de multiplicação de matrizes** (*matriz A x matriz B = matriz C*) é o caso de estudo sobre o qual incide este trabalho prático, uma vez que é o problema ideal para estudar a variação da performance do processador quando se trabalha com acessos a grandes conjuntos de dados guardados em memória.

Neste problema, torna-se relevante perceber a organização dos dados em memória e como tomar partido desta de forma a melhorar a performance (os dados podem estar guardados em locais de memória com acesso mais rápido, que permitem melhorar significativamente a performance para casos de teste com valores mais elevados, i.e. acessos a memória cache L1 são mais curtos do que acessos à memória principal). Para tal, foram implementados diferentes algoritmos que evidenciam a utilização de recursos do processador para cada uma dessas variações.

Descrição dos algoritmos implementados

1. Multiplicação Normal

Algoritmo simples e intuitivo de multiplicação de matrizes. Consiste em percorrer a linha `i` da primeira matriz *A* e a coluna `j` da segunda matriz *B* e multiplicar os elementos da primeira pelos elementos correspondentes da segunda, somando o resultado na posição `(i,j)` de uma terceira matriz *C*, inicialmente nula, como representado no excerto seguinte. Este algoritmo foi implementado em *C++* e em *Java*.

```
1    for (i=0; i<n; i++)
2      for (j=0; j<n; j++)
3        for (k=0; k<n; k++)
4          phc[i*n+j] += pha[i*n+k]*phb[k*n+j];
```

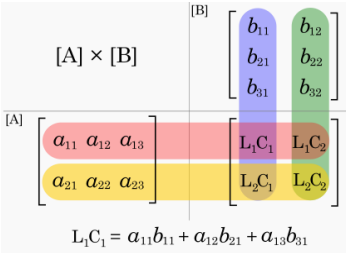


Figura 1. Excerto de Código e Esquema do Algoritmo da Multiplicação Normal

2. Multiplicação por Linha

O algoritmo anterior foi alterado para permitir, teoricamente, que nesta segunda estratégia sejam aproveitados mais dados que se encontram em memórias mais rápidas, de forma a melhorar a performance em relação à primeira resolução do problema. Para aumentar a quantidade de acertos no acesso a memória cache era ideal a segunda matriz ser iterada também por linhas, devido à forma como um processador guarda os valores em cache (ao guardar em cache uma entrada de uma matriz, guarda também as entradas seguintes por se encontrarem em posições de memória consecutivas).

Neste algoritmo pretende-se fixar um elemento da matriz *A*, por exemplo `(i,k)`, e fazer todas as operações relacionadas com este, percorrendo toda a linha `k` da matriz *B* e somando o seu resultado ao elemento `(i,j)` da matriz *C*, onde `j` é a coluna correspondente ao elemento da matriz *B* multiplicado. Neste processo é mais uma vez importante inicializar todos os elementos da matriz *C* a 0 de forma a servirem como acumulador dos valores intermediários do cálculo. Este algoritmo foi implementado em *C++* e em *Java*.

```
1    for (i=0; i<n; i++)
2      for (k=0; k<n; k++)
3        for (j=0; j<n; j++)
4          phc[i*n+j] += pha[i*n+k]*phb[k*n+j];
```

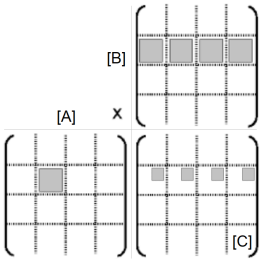


Figura 2. Excerto de Código e Esquema do Algoritmo da Multiplicação por Linha

3. Multiplicação em Bloco

O algoritmo anterior poderia perder o seu benefício de guardar em memórias de acesso rápido toda uma linha da matriz *B* (ao ser lido um bloco de memória e não apenas o elemento pretendido) quando estas linhas são maiores do que o tamanho do bloco copiado. Para ultrapassar essa situação, a multiplicação das matrizes *A* e *B* foi realizada dividindo estas em nBk^2 submatrizes (blocos) de tamanho $bkSz$. Os blocos são percorridos, através dos três primeiros ciclos da figura 3, e é-lhes aplicado o algoritmo 2 para realizar a multiplicação das submatrizes, que são acumuladas no bloco correspondente da matriz *C*. Este algoritmo foi implementado apenas em *C++*.

```
1.   for(i = 0; i < nBk; i++)
2.    for(j = 0; j < nBk; j++)
3.     for(k = 0; k < nBk; k++)
4.
5.      for(l = 0; l < bkSz; l++)
6.       for(t = 0; t < bkSz; t++)
7.        for(c = 0; c < bkSz; c++)
8.          phc[(i*bkSz+l)*n + (j*bkSz+c)]
               += pha[(i*bkSz+l)*n + (k*bkSz+t)]
                * phb[(k*bkSz+t)*n + (j*bkSz+c)];
```

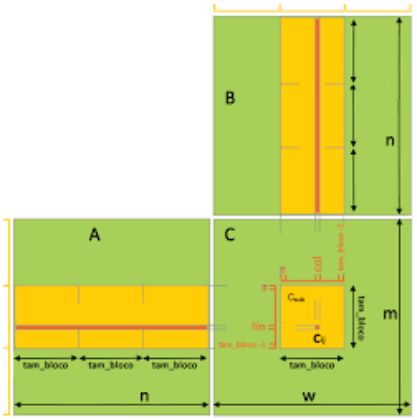


Figura 3. Excerto de Código e Esquema do Algoritmo da Multiplicação em Bloco

Medidores de performance

Existem inúmeros medidores diferentes para quantificar a performance do processador do computador, mas só é disponibilizado, em qualquer momento, acesso a no máximo 10 contatores (limite imposto pelo hardware do ambiente de teste) que sejam necessariamente compatíveis entre si.

De modo a obter uma melhor perspetiva da variação de performance do processador para as versões dos algoritmos implementados, em C++, foram selecionados e utilizados os seguintes contadores:

- *PAPI_L1_DCM*: contador para a número de falhas de cache de nível 1;
- *PAPI_L2_DCM*: contador para a número de falhas de cache de nível 2;
- *PAPI_TOT_CYC*: contador do número total de ciclos realizados;
- *PAPI_TOT_INS*: contador do número total de instruções completas.

Além das medidas obtidas pelos contadores, foi ainda calculado o tempo de execução dos casos de teste (***TOT_TIME***) e, em prol de uma melhor compreensão dos dados obtidos, foram derivados alguns medidores extra da performance do processador a partir dos acima enunciados. O número de **Ciclos por Instrução** em (1), o número de **Operações de Vírgula Flutuante por Segundo** em (2) e a **soma de falhas na cache de nível 1 e nível 2** em (3).

$$CPI = \frac{PAPI\_TOT\_CYC}{PAPI\_TOT\_INS} \quad (1)$$

$$GFlops = \frac{COMPLEXITY}{TOT\_TIME} \cdot 10^{-9} \quad (2)$$

$$L1\&L2\_TOT\_MISS = PAPI\_L1\_DCM + PAPI\_L2\_DCM \quad (3)$$

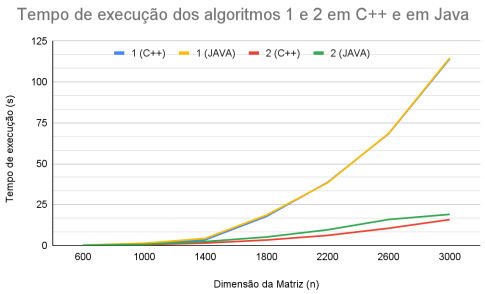
Os medidores de performance disponibilizados pelo software *PAPI* acima selecionados são os medidores necessários e suficientes para obter os derivados apresentados, sendo estes os mais relevantes na análise comparativa, uma vez que representam mais fidedignamente a performance do *core*. Além disso, são medidas universais cujo significado e valor já é reconhecido universalmente.

No cálculo dos *GFlops* de cada um dos testes realizados a complexidade de todos os algoritmos implementados foi aproximada a $O(n^3)$, uma vez que esta simplificação em termos comparativos não afeta a análise nem veracidade dos resultados.

Resultados Obtidos e Análise

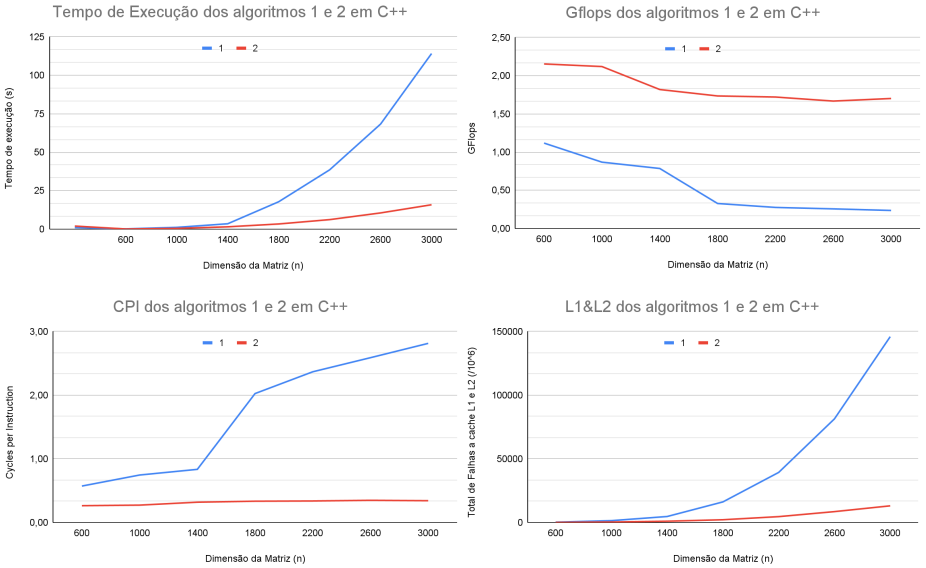
Com o objetivo de avaliar e comparar as versões do algoritmos implementados de forma fidedigna, foi usado, durante toda a avaliação, o mesmo ambiente de teste, o computador do laboratório, de forma a que diferentes características do hardware não afetassem o rigor e veracidade dos resultados. Cada teste foi realizado 3 vezes (exceto para o algoritmo de multiplicação por bloco), aumentando assim o poder estatístico uma vez que foi aplicada a sua média. Foi analisada, igualmente, a performance para a linguagem *JAVA* no primeiro e segundo algoritmos. Os dados foram posteriormente tratados culminando nos seguintes gráficos.

Analisando o gráfico abaixo, é evidente que as linguagens *C++* e Java são semelhantes a nível de performance, apresentando ambos o mesmo padrão de escalabilidade ainda que o *C++* seja ligeiramente mais rápido, principalmente no 2º algoritmo.

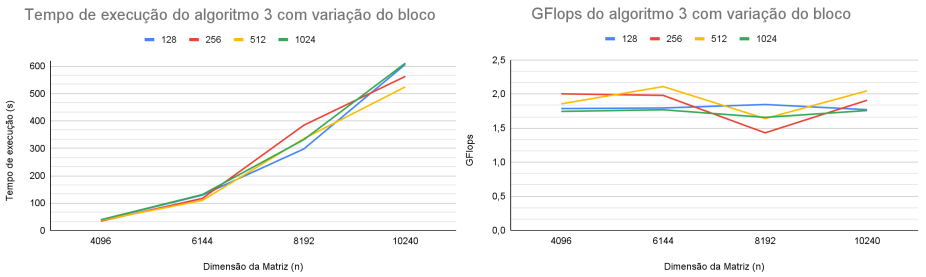


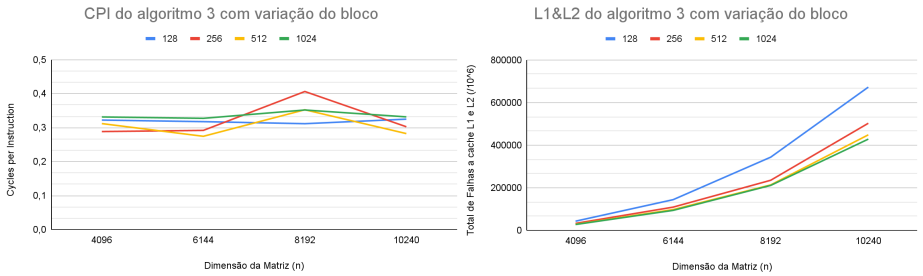
De seguida foram comparados os medidores de performance obtidos do algoritmo 1 e 2 (apenas para *C++*) e é possível observar que o algoritmo 2 apresenta um indicador de *Gflops* superior e mais constante para dimensões das matrizes superiores. Isto faz sentido uma vez que falhas de cache

reduziram significativamente, ou seja, o tempo é menos utilizado para recuperar valores guardados em memória e mais para efetuar as operações sobre esses mesmos valores. Assim, com apenas uma pequena alteração no código, foi possível obter uma performance bastante melhor no algoritmo 2 relativamente ao algoritmo 1, uma vez que é o que tira mais proveito da organização dos dados em memória diminuindo o número de falhas de cache.

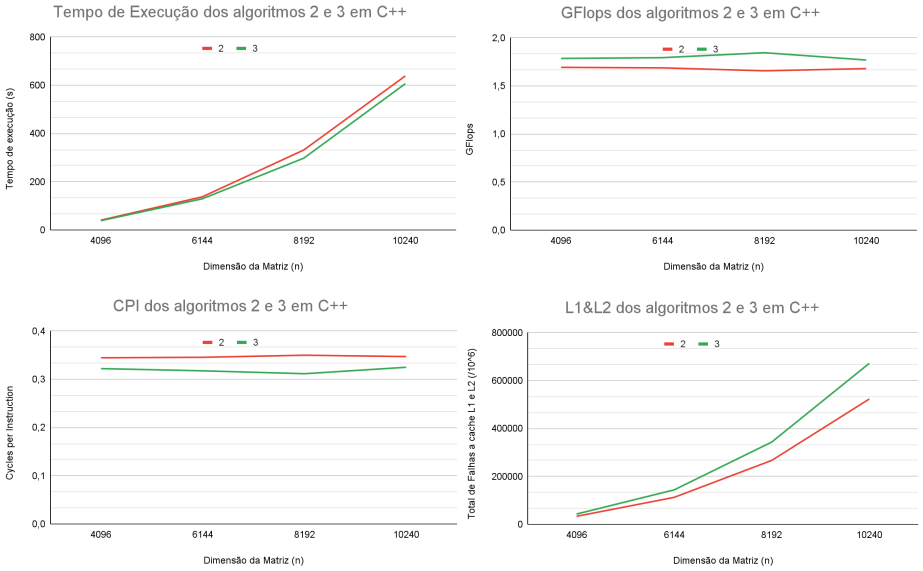


Observando os dados obtidos para diferentes tamanhos de bloco, na aplicação do algoritmo 3, não é claro qual é o que possui uma melhor performance. Ainda assim, é possível observar que os testes com blocos de 256 e 512 apresentaram amplas oscilações ao nível das operações em vírgula flutuante por segundo e de *CPI*, quando se esperava um comportamento mais constante, semelhante ao dos blocos de tamanho 128 e 1024. Além disso, observa-se que as falhas de acesso à memória cache *L1&L2* apresentaram um comportamento esperado, uma vez que a multiplicação com blocos de 1024 apresenta valores menores do que com os blocos de 128. Este é um valor previsível pois com maiores blocos reduz-se a necessidade do processador alterar a memória guardada em cache, até a um valor máximo, que aparenta estar próximo de 1024 já que a diferença da curva *L1&L2* entre dois tamanhos de blocos de teste consecutivos é mais insignificante com o aumento do tamanho destes (curvas para 512 e 1024 já se encontram quase sobrepostas). Pode-se considerar o tamanho de bloco de 128 por 128 como sendo o mais consistente e por essa mesma razão será utilizado para comparar com algoritmo de multiplicação em linha.





Juntando os resultados obtidos para o algoritmo de multiplicação em linha e o algoritmo de multiplicação em bloco, com blocos de 128, obtém-se os gráficos abaixo. Os resultados de ambos os algoritmos são bastante parecidos, tendo o algoritmo de multiplicação em linha melhor performance apenas quando comparamos a soma das falhas *L1&L2*. Estes resultados não são surpreendentes, pois o algoritmo de multiplicação em bloco foi desenvolvido para ser um refinamento do algoritmo de multiplicação em linha.



Conclusão

Após a análise dos resultados da multiplicação de matrizes é possível afirmar que a multiplicação de matrizes normal é muito ineficiente quando comparada a outros algoritmos, característica evidenciada tanto no tempo de execução como no número de falhas de cache. Por outro lado, pode-se concluir que há pouca diferença entre o algoritmo de multiplicação por linha e o algoritmo de multiplicação em bloco quando a multiplicação das matrizes do algoritmo dos blocos é também realizada com a estratégia de multiplicação em linha.

Assim, num mundo onde todo o tempo e recursos contam, realça-se a importância da construção de algoritmos que tirem o máximo proveito da organização dos dados em memória, em prol da eficiência e melhoria da performance dos mesmos.

Anexos

Tabela dos resultados recolhidos e anexo para o *Google Sheets* usado no seu tratamento.

Assign 1 - Data Analysis

Method	MatrixSize	BlockSize	Duration	PAPI_L1_DCM	PAPI_L2_DCM	PAPI_TOT_CYC	PAPI_TOT_INS
1	600	0	0.194519	244526955	39879262	874940883	1518177175
1	600	0	0.192347	244552859	40953366	866946723	1518173947
1	600	0	0.192555	244544909	39172762	867334994	1518173940
1	1000	0	1.14024	1223971734	241349354	5201476625	7017064189
1	1000	0	1.15651	1222275939	258087664	5248098725	7017064853
1	1000	0	1.156	1222300610	251138702	5258855734	7017064834
1	1400	0	3.38168	3416714791	1042459699	15540638018	19241397724
1	1400	0	3.46402	3532343919	1135282115	15979151718	19241398366
1	1400	0	3.63605	3532331282	1520631731	16701177492	19241398408
1	1800	0	17.7502	9091488244	6549832002	82590095513	40879178426
1	1800	0	17.8762	9061674679	7561652482	83202685884	40879179101
1	1800	0	17.7749	9061725391	7258539516	82675557475	40879179086
1	2200	0	38.912	17621083224	21502381767	175925020187	74618400302
1	2200	0	38.4103	17622227392	21850560389	176682535871	74618400158
1	2200	0	38.3499	17622398650	21874155073	177531566997	74618400194
1	2600	0	68.6243	30882436246	50924738852	320354078062	123147066307
1	2600	0	68.2235	30882397634	50464513424	318380053652	123147066207
1	2600	0	68.1561	30882515410	50245651449	318009531198	123147066164
1	3000	0	114.507	50304846588	96065178075	533284640705	189153178255
1	3000	0	114.076	50305090748	95659814029	532389828421	189153178124
1	3000	0	113.887	50305255559	95319332172	531444508913	189153178069
2	600	0	0.100531	27124566	56935069	459612413	1733095091
2	600	0	0.100829	27230924	57289839	458344257	1733092516
2	600	0	0.099417	27234257	57298769	456958441	1733092440
2	1000	0	0.471385	125846853	261898008	2192016252	8014064404
2	1000	0	0.471551	126063119	262211601	2191595985	8014060362
2	1000	0	0.471793	126065476	262495569	2188135312	8014060348

2	1400	0	1.52302	346453177	676502944	7042108165	21979514528
2	1400	0	1.49988	346791930	680658386	7005887922	21979508601
2	1400	0	1.50103	346793714	681410212	7012212511	21979508578
2	1800	0	3.36443	745820468	1437056257	15662021131	46701446606
2	1800	0	3.35209	745317919	1443288520	15621777304	46701458225
2	1800	0	3.3396	745315072	1442180241	15621171218	46701458247
2	2200	0	6.18426	2075407614	2551147938	28823132946	85251875669
2	2200	0	6.17292	2075582844	2555990412	28836933979	85251875688
2	2200	0	6.19923	2075463464	2552538778	28912368212	85251875671
2	2600	0	10.2966	4413868195	4165196443	48065425358	140702776880
2	2600	0	10.2997	4413833386	4155053400	48064225289	140702776891
2	2600	0	11.0419	4413767370	4125968765	51093507738	140702777037
2	3000	0	15.8717	6781959503	6328509456	74025206564	216126159293
2	3000	0	15.849	6781953300	6332904268	74016480588	216126159270
2	3000	0	15.8614	6781955220	6323470336	74062343427	216126159265
2	4096	0	40.5619	17536506337	15958000079	189470044066	549990929982
2	4096	0	40.6235	17537458623	15948858585	189673758592	549990929984
2	4096	0	40.5493	17534372727	15929967106	189420141004	549990929962
2	6144	0	137.334	59120970890	53420467903	641354674448	1855954777738
2	6144	0	137.484	59123305112	53464110168	641297725085	1855954777802
2	6144	0	137.34	59123757037	53456085140	641567417138	1855954777771
2	8192	0	331.039	140039148714	126718120130	1535097777656	4398986720560
2	8192	0	337.688	140038489824	126881187325	1558050362453	4398986722262
2	8192	0	326.514	140021270356	126756232866	1524726614185	4398986719368
2	10240	0	638.857	273374949715	249768737232	2981902078998	8591403622633
2	10240	0	639.418	273373419615	249912217534	2985024992256	8591403622788
2	10240	0	638.45	273372587761	250135408243	2980101963817	8591403622481
3	4096	128	38.4738	9726433912	33229508140	178545417197	554223650053
3	6144	128	129.239	32839851492	111066091679	593954422158	1870353496908
3	8192	128	297.844	77892015611	265602396029	1381179576766	4433251274748
3	10240	128	606.69	151929461867	520348705188	2813332454594	8658483908482
3	4096	256	34.3165	9091236618	23552322638	159202632505	552047233805

3	6144	256	117.17	30675218824	78231846977	543652847200	1863008121519
3	8192	256	384.395	72953908287	162089908107	1794168989187	4415840066574
3	10240	256	562.695	142026442172	360605605265	2608913329898	8624477602658
3	4096	512	37.0345	8763818903	19616981180	171810562485	550966335928
3	6144	512	109.972	29622872428	67676227407	510260155898	1859360091247
3	8192	512	335.355	70202773304	143013382645	1553723973110	4407192876111
3	10240	512	524.557	136959400281	311343953498	2430884571013	8607588587297
3	4096	1024	39.3887	8805565961	18537297642	182413522849	550427686110
3	6144	1024	131.15	29779834403	63874560994	608051397228	1857542151757
3	8192	1024	331.793	70594580617	140385266886	1549232080871	4402883677055
3	10240	1024	610.937	137920452664	290267122169	2851534924468	8599172200732