Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Projeto 1 - 'Performance evaluation of a single core'

Trabalho realizado por
Grupo 2:
Pedro Alves (up201707234)
Jorge Pacheco (up201705754)
Raul Viana (up201208089)

18 de março de 2021

Table of Contents

Introdução

A programação paralela, apesar de trazer grandes benefícios em tempo de processamento, deve ser aplicada em programas que estejam bem desenhados e escritos. Isto é, antes de se partir para a paralelização de um programa é necessário garantir que este se encontra otimizado. Nas linguagem compiladas o compilador realiza algumas otimizações orientadas ao funcionamento intrínseco do processador, no entanto, devido às limitações do processo de compilação, esse também é um papel importante do programador.

Assim, neste trabalho pretende-se demonstrar a importância da otimização do código, de forma a que corra no processador com o menor número possível de falhas de cache. Para isso irá ser otimizado um algoritmo de multiplicação de matrizes, garantindo assim que o programa pode escalar para matrizes extremamente grandes sem um aumento exponencial do tempo de processamento. Estudar-se-á também se a otimização é inerente ao funcionamento do processador, utilizando para isso a implementação do algoritmo em duas linguagens diferentes.

Para a medição dos tempos de processamento foi implementado um programa que corre o algoritmo de multiplicação diversas vezes, de forma contínua, para os tamanhos de matriz desejados e que grava num ficheiro “.csv” as variáveis em análise, tornando dessa forma mais fácil o seu tratamento posterior. A partir do algoritmo base fornecido, que multiplica as matrizes utilizando exatamente o método algébrico, foram desenvolvidos outros dois mais eficientes. O segundo multiplica as matrizes linha a linha e o terceiro que utiliza o mesmo método mas em sub-matrizes, mais pequenas, da matriz fornecida.

Foram utilizadas as linguagens de programação C++ e Java para implementar os vários algoritmos de multiplicação de matrizes. É de especial importância notar que, no que toca ao armazenamento em memória de arrays multi-dimensionais, estas linguagens diferem da seguinte forma: enquanto que em C++ estes arrays são armazenados em “row-major order”, ou seja, tanto os elementos que compõe uma linha de uma matriz como as linhas em si são armazenadas em posições de memória contíguas; em Java um array multi-dimensional não passa de um array composto por várias referências para outros arrays, estando estes armazenados em blocos de memória totalmente não relacionados.

A recolha de dados foi realizada num processador Intel, i3-8100, com uma frequência máxima de 3.6GHz e 6MB de cache em três níveis.

Algoritmo standard

O algoritmo standard para a multiplicação de matrizes é um algoritmo que simula a multiplicação algébrica manual de duas matrizes. Multiplica cada elemento de uma linha de uma matriz por cada elemento correspondente da coluna da outra matriz, somando todas essas multiplicações. Na figura seguinte é possível observar as primeira quatro iterações desse algoritmo.

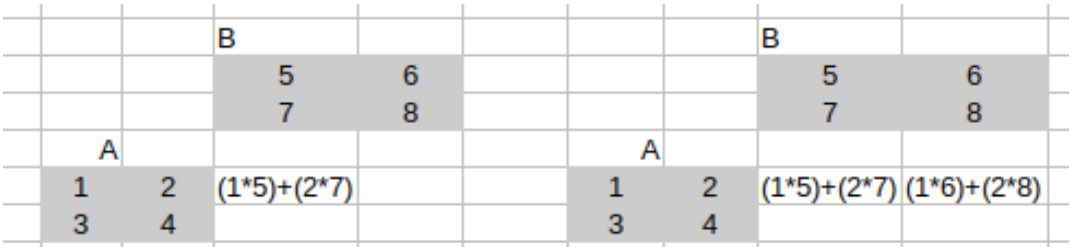


Figura 1: demonstração das quatro primeiras iterações do algoritmo standard

Como é possível observar na figura 1 o algoritmo percorre a linha da matriz A enquanto percorre também a coluna da matriz B para efetuar o calculo de cada célula na matriz final.

Foi desenvolvido o mesmo algoritmo, com o mesmo comportamento, também na linguagem Java. Desta forma pretendeu-se avaliar se a evolução do desempenho de processamento em função do tamanho da matriz se deve a aspetos da arquitetura interna do processador ou a aspetos intrínsecos da linguagem C++.

O gráfico seguinte apresenta os resultados obtidos.

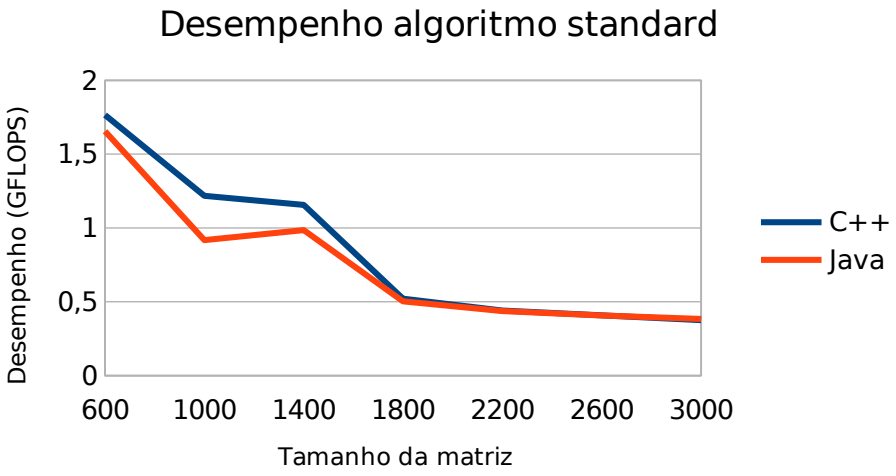


Gráfico 1: tempo de processamento em função do tamanho da matriz – algoritmo standard em C++ vs Java

O gráfico 1 demonstra que este algoritmo não é propriamente eficiente para um tamanho de matrizes relativamente grande, e que não permite escalar o problema de forma conveniente.

A comparação da performance entre as implementações nas duas linguagens permite observar que o comportamento do tempo de processamento é muito semelhante, com ligeiras diferenças, que podem ser explicadas por variações introduzidas pelo escalonamento eventual de outros processos.

Algoritmo linha por linha

Uma implementação mais eficiente do algoritmo de multiplicação de matrizes é o algoritmo linha por linha. Este algoritmo não segue os passos realizados na multiplicação algébrica, de forma a aproveitar a arquitetura e forma de funcionamento do processador.

Assim, cada elemento da primeiro matriz é multiplicado por cada elemento da linha correspondente da segunda matriz e os resultados vão sendo colocados na linha correspondente da matriz resultante. Na figura seguinte é possível observar as primeiras quatro iterações desse algoritmo.

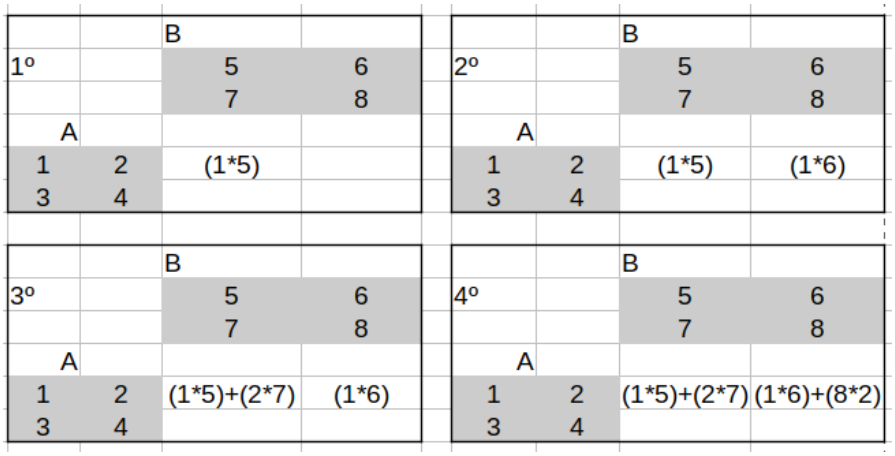


Figura 2 – demonstração das 4 primeiras iterações do algoritmo linha por linha

Pela análise da figura 2 é possível observar que o algoritmo linha por linha tira todo o partido das características de funcionamento desenvolvidas para a cache do processador: as localidades espacial e temporal. Quando algum elemento da memória é necessário para o processamento, é carregado para a cache uma linha com vários elementos. Isto porque estatisticamente quando um elemento é necessário existe uma grande probabilidade de os elementos próximos serem também necessários nos próximos ciclos – localidade espacial. Por outro lado, esses elementos são mantidos na cache até haver necessidade de os substituir,pois estatisticamente um elemento que tenha sido necessário será muito provavelmente necessário novamente nos próximos ciclos – localidade temporal.

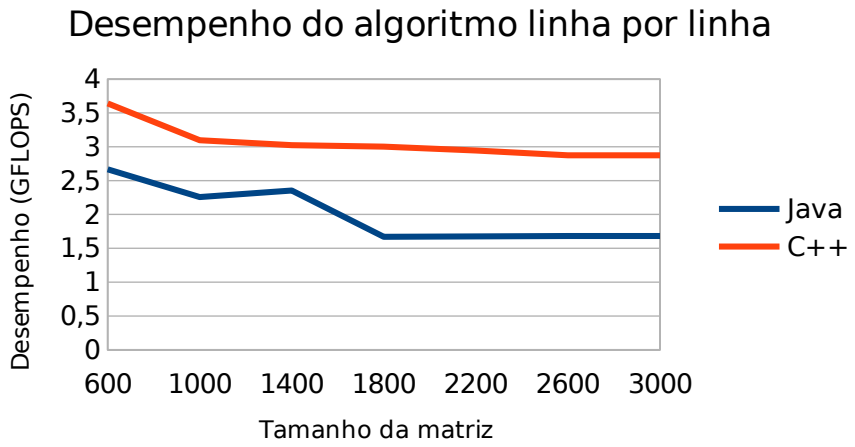
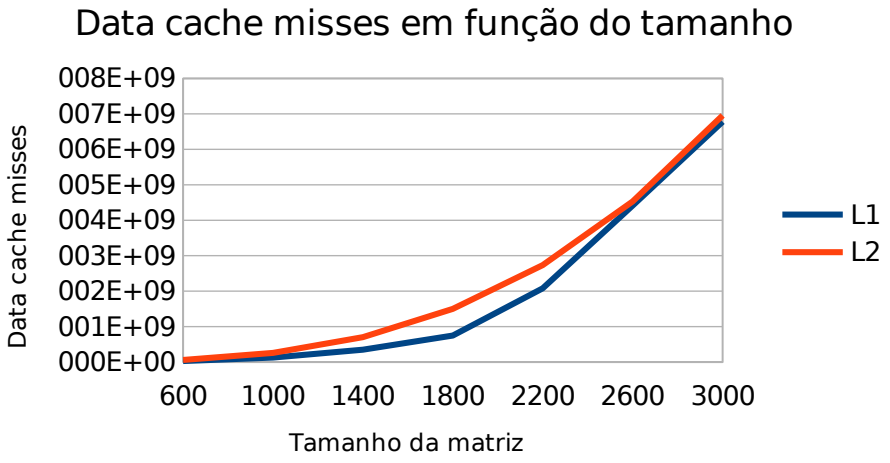


Gráfico 2: tempo de processamento em função do tamanho da matriz – C++ vs Java

A implementação deste algoritmo foi também efetuada em duas linguagens diferentes, e no gráfico 2 encontram-se os valores de desempenho medidos para diferentes tamanhos de matrizes nas duas linguagens.

A análise do gráfico 2 e a comparação com o gráfico 1 permitem observar algumas diferenças entre os algoritmos standard e linha por linha.

Na comparação entre a performance das implementações nas duas linguagens é possível observar que as duas implementações registam uma diferença média de cerca de 3 vezes no desempenho. O algoritmo de multiplicação linha por linha tem um desempenho bastante superior ao algoritmo standard. Este aumento de performance deve-se ao número de falhas de dados na memória cache no processador. Os dados encontram-se disponíveis na cache do processador em blocos. Esses blocos representam as linhas, ou parte delas das matrizes. Assim, um algoritmo que tire partido da leitura sequencial das linhas das matrizes torna-se muito mais eficiente, sendo este o caso do algoritmo linha por linha.



Gráfio 3: número de falhas de dados de cache em função do tamanho da matriz

No gráfico 3 é possível observar o comportamento do número de falhas da dados na cache do processador à medida que o tamanho da matriz vai aumentando, na implementação em C++. São estas falhas que levam a que o tempo de processamento cresça exponencialmente. No gráfico 4 é possível observar a comparação entre o desempenho da cache nos algoritmos standard e linha por linha. Os valores de "data cache misses" ("L1 – Std" e "L2 – Std") no algoritmo standard começam a ter um aumento exponencial a partir de matrizes de tamanho 1800, enquanto que no algoritmo linha por linha esse aumento se mantém aproximadamente linear.

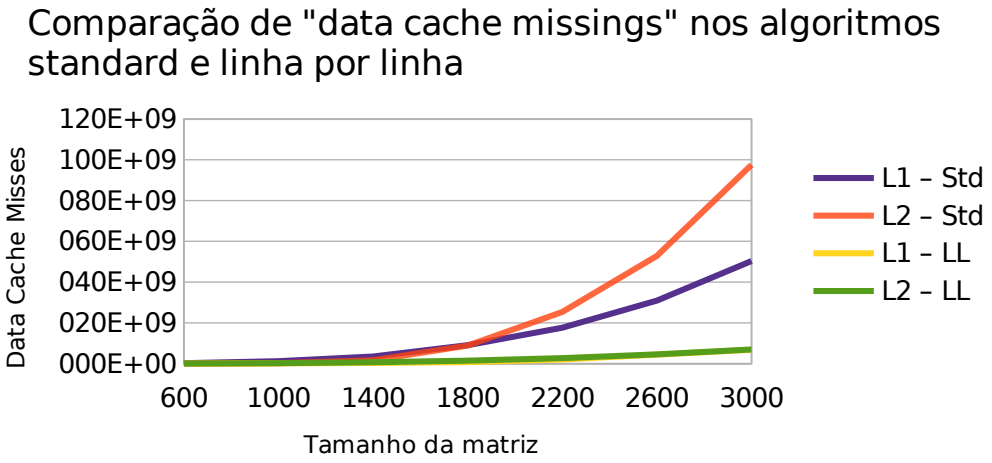


Gráfico 4: data cache misses nos algoritmos standard e linha por linha

Sempre que a informação não se encontra na cache (data cache miss) esta tem de ser carregada da memória principal, o que é uma operação de I/O com um custo muito elevado de ciclos de relógio. Assim a diferença de performance entre estes dois algoritmos prende-se com a eficiência da gestão e acesso dos dados à memória cache do processador.

No entanto, mesmo o algoritmo linha por linha tem as suas limitações de escalabilidade, uma vez que a matriz pode ser tão grande que uma linha não cabe na memória cache, o que vai levar ao problema do algoritmo standard, a necessidade de carregar diversas vezes os dados da memória principal.

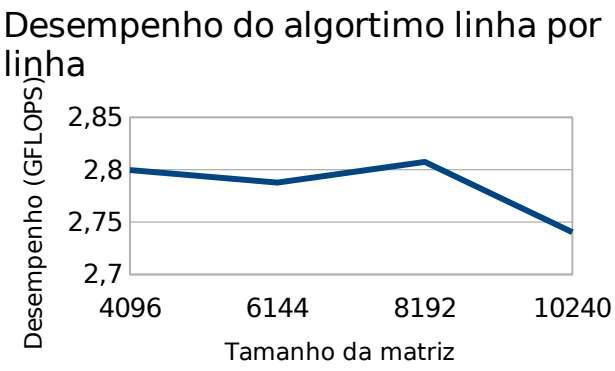


Gráfico 5

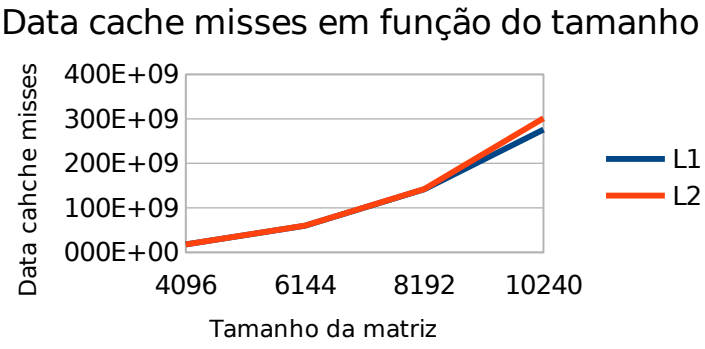


Gráfico 6

Assim, os gráficos 5 e 6, mostram que a partir de matrizes de tamanhos 8192 o desempenho no processamento para o algoritmo linha por linha também começa a diminuir consideravelmente, devido ao aumento de “data cache misses”, principalmente da cache L2.

Algoritmo por blocos

Uma possível forma de contornar este problema e manter a escalabilidade da multiplicação de matrizes seria implementar um algoritmo que dividisse a linha em partes que pudessem ser contidas na cache. Desta forma foi implementado o algoritmo por blocos, que divide a matriz em blocos mais pequenos, efetuando os cálculos contidos dentro dessa sub-matriz, levando a que a multiplicação possa ser escalada para matrizes de tamanhos muito grandes e que a performance seja afinada, encontrando um tamanho de bloco ideal para o processador em utilização.

Foi registado o tempo de processamento para diferentes tamanhos de matrizes e blocos. Os resultados encontram-se no gráfico 7.

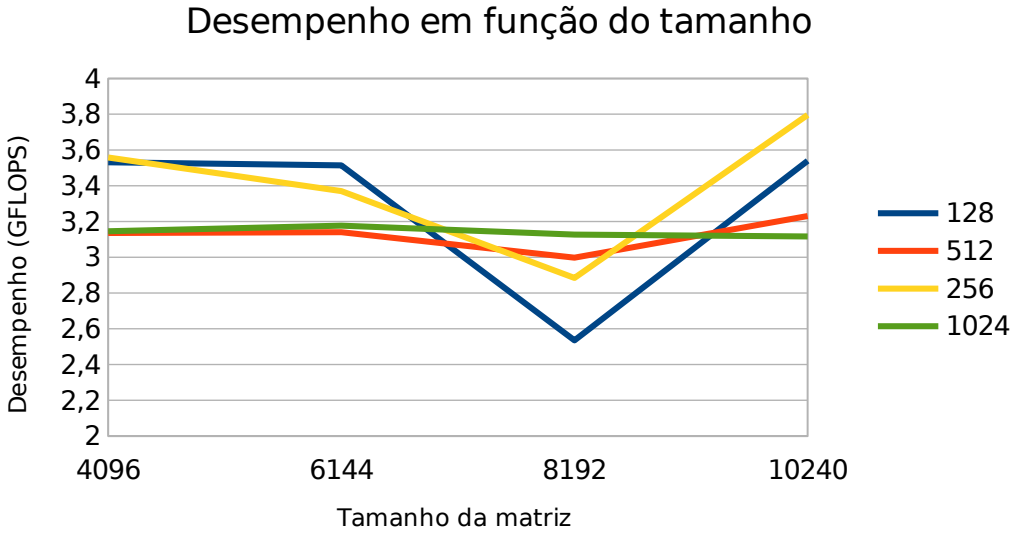


Gráfico 7: diferentes tamanhos de matrizes e blocos

Através do gráfico 7 é possível observar que o tamanho ótimo do bloco para este processador é de 256. De notar uma quebra de desempenho para o tamanho de 8192, principalmente para os blocos de menores dimensões. No entanto o gráfico sugere que para tamanhos superiores a 10240 o desempenho tende a melhorar principalmente para blocos de 256.

Comparando o desempenho de processamento dos algoritmos linha por linha e por blocos, no gráfico 8, é possível observar que o algoritmo por blocos é mais eficiente, apesar de não apresentar um aumento de performance tão acentuado como seria inicialmente expectável. Foi utilizado o tempo de processamento com o tamanho de bloco de 256, por ter sido considerado o tamanho de bloco ótimo.

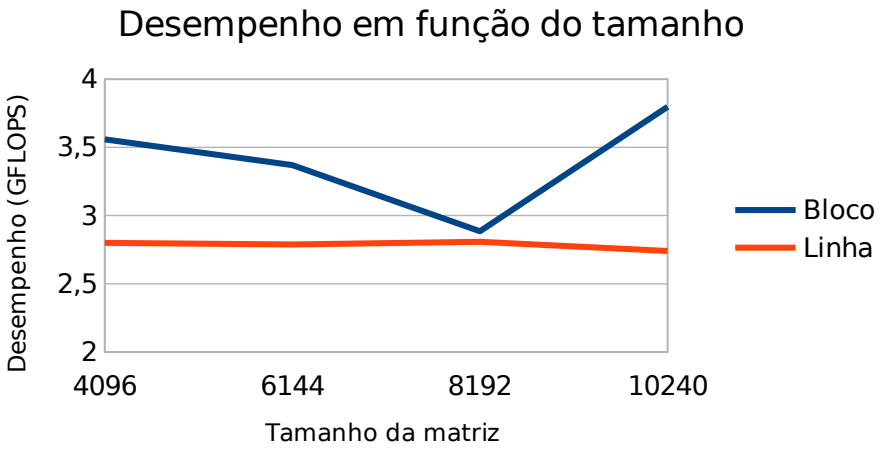


Gráfico 8: Tempo de processamento dos algoritmos por linha e por bloco

A partir de matrizes de tamanho 8192 o algoritmo linha por blocos começa a apresentar um aumento aumento do desempenho em relação ao algoritmo linha por linha. Isto deve-se provavelmente por ser a partir deste tamanho de matriz que a linha não cabe toda na memória cache, o que provoca um maior número de “data cache misses”, o que leva à degradação do desempenho.

Este comportamento poderá estar relacionado com a quantidade de “data cache misses”. Como o tamanho da matriz se torna extremamente grande, cada linha tem de ser carregada por partes, o que limita o desempenho do algoritmo por linha.

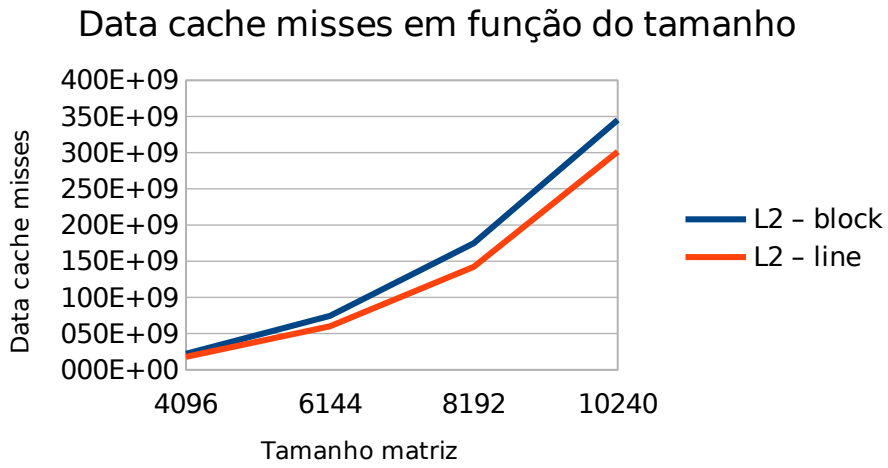


Gráfico 9: data cache misses nos algoritmo por linha e por bloco

No gráfico 9 é possível observar a diferença no número de data cache misses para matrizes de tamanho superior a 8192, entre os dois algoritmos, que explica a diferença no desempenho de processamento.

Conclusão

A multiplicação de matrizes pode ser modelada de forma diferente da forma como é realizada algebricamente. Essa mudança leva a aumentos significativos na performance do algoritmo.

A otimização do algoritmo deve ser realizada tendo em conta o funcionamento interno do processador, nomeadamente a localização espacial e temporal da cache.

Qualquer processo de otimização de código, com vista à introdução de paralelismo deve ser sempre precedida por um estudo da otimização do código em série, pois na grande maioria das vezes é possível obter ganhos significativos.

Anexo 1 – dados recolhidos

Neste anexo encontram os dados recolhidos durante o processamento dos vários algoritmos com os diferentes inputs requeridos.

Agoritmo standard em C++

Size	Time	L1	L2
600	0,24477	244806664	42692166
1000	1,64234	1224650781	256217363
1400	4,74684	3520341095	1700444516
1800	22,386	9059253458	8873232145
2200	48,3412	17644124045	25375241449
2600	86,2234	30912266518	52844144647
3000	143,751	50317568625	97470458573

Algoritmo standard em Java

Size	Time
600	0,261
1000	2,179
1400	5,568
1800	23,187
2200	48,726
2600	86,293
3000	140,909

Algoritmo linha por linha em C++

Size	Time	L1	L2
600	0,118653	27109212	56647726
1000	0,645767	125729167	257837400
1400	1,81505	346028960	700519840
1800	3,88589	745244062	1496961085
2200	7,23302	2077942527	2733410635
2600	12,2297	4412815940	4528115554
3000	18,7943	6779524738	6953583334

Algoritmo linha por linha em Java

Size	Time
600	0,162
1000	0,886
1400	2,331
1800	6,987
2200	12,716
2600	20,915
3000	32,138

Algoritmo linha por linha em C++ em matrizes de muito grandes dimensões

Size	Time	L1	L2
4096	49,092	17699318156	17737250883
6144	166,394	59715703759	59963715709
8192	391,648	141502313959	142239994261
10240	783,672	275917139510	301361384704

Algoritmo por blocos em C++

	Size	Time	L1	L2
128	4096	38,9218	9768533509	31791024389
	6144	132,014	32951401543	106939697051
	8192	433,762	78535041396	245225413128
	10240	606,687	152661225018	492087545430
256	4096	38,627	9096762523	22036705046
	6144	137,654	30698136544	74265420505
	8192	381,152	72783669879	174897592982
	10240	565,703	142014943557	345218634284
512	4096	43,8286	8772049792	19187230595
	6144	147,725	29637195827	65051423715
	8192	366,784	70283301001	151823433853
	10240	664,725	137060077019	301784226568
1024	4096	43,7002	8794450565	18134155665
	6144	146,009	29736627681	61513114591
	8192	351,694	70491236349	144714141044
	10240	689,095	137696096108	282728755849

Desempenho Algoritmo Standard – C++ vs Java

	GFLOPS	
	C++	**Java**
600	1,76492217183478	1,6551724137931
1000	1,21777463862538	0,917852225791648
1400	1,15613755677461	0,985632183908046
1800	0,521039935674082	0,50304049683012
2200	0,440535195650915	0,437056191766203
2600	0,407685152754357	0,407356332495104
3000	0,375649560698708	0,383226053694228

Desempenho algoritmo linha por linha

	GFLOPS	
	C++	**Java**
600	3,64086875173826	2,66666666666667
1000	3,09709229489893	2,25733634311512
1400	3,02360816506432	2,35435435435435
1800	3,00162897045465	1,66938600257621
2200	2,94427500546107	1,67474048442907
2600	2,87431416960351	1,68070762610567
3000	2,87321155882369	1,68025390503454

	GFLOPS
	C++
4096	2,79962017175915
6144	2,78769948416409
8192	2,80739752986355
10240	2,74028375136537

Desempenho algoritmo por blocos

	GFLOPS			
	128	**256**	**512**	**1024**
4096	3,53115615	3,55810582	3,13582805	3,14504175
6144	3,51369149	3,36972749	3,13999978	3,17690326
8192	2,53482700	2,88470644	2,99770881	3,12633035
10240	3,53968957	3,79613268	3,23063470	3,11638257