U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

1º TRABALHO LABORATORIAL

Avaliação de desempenho de uma implementação *single-core* e de uma implementação *multi-core*

Unidade Curricular: Computação Paralela e Distribuída (LEIC.028)

Ano Letivo: 2024/2025

Professor da Componente Prática: Vítor Rodrigues

Turma: 3LEIC06

Grupo: 11

Estudantes e Autores:

Beatriz Pereira up202207380@fe.up.pt

Luana Lima up202206845@fe.up.pt

Margarida Fonseca up202207742@fe.up.pt

Índice

1. Descrição do Problema

Neste projeto exploramos o problema da multiplicação de matrizes e a sua influência no desempenho dos processadores, algo bastante estudado devido à complexidade computacional.

A multiplicação de matrizes quadradas tem uma complexidade temporal de $O(N^3)$, onde N representa a sua dimensão. Contudo, diferentes abordagens podem influenciar significativamente o desempenho deste cálculo, especialmente no que diz respeito ao acesso à memória e à eficiência da cache.

O principal objetivo deste trabalho é compreender que implementações são mais *cache-friendly* e que técnicas podem ser aplicadas para otimizar o desempenho da multiplicação de matrizes.

2. Explicação dos Algoritmos

Neste projeto, implementamos, em duas linguagens de programação diferentes (C++ e Java), três algoritmos distintos de multiplicação de matrizes com o objetivo de estudar o impacto da hierarquia de memória no desempenho desta operação. Implementamos também duas versões paralelizadas da multiplicação por linhas, utilizando OpenMP para verificar o impacto do paralelismo na melhoria do desempenho. Adicionalmente, desenvolvemos para os diferentes métodos, a multiplicação de matrizes não quadradas, aumentando a generalidade dos testes.

2.1. Multiplicação de Matrizes Simples

O primeiro algoritmo segue a definição matemática clássica da multiplicação de matrizes. Para cada posição ***(i, j)*** da matriz resultante, multiplica-se a linha ***i*** da primeira matriz pela coluna ***j*** da segunda. Este método é de fácil implementação mas é pouco eficiente em termos de acesso à memória, uma vez que implica múltiplos saltos entre diferentes localizações da memória devido ao acesso a colunas da segunda matriz, podendo resultar num elevado número de cache misses.

2.2. Multiplicação de Matrizes por Linha

Este algoritmo melhora a abordagem anterior reorganizando a ordem das operações para otimizar o acesso à memória. Em vez de multiplicarmos cada linha por coluna, realizamos a multiplicação linha a linha, ou seja, cada elemento da linha da primeira matriz é multiplicado por toda a linha correspondente da segunda matriz, acumulando os valores diretamente na matriz resultante. Esta abordagem melhora significativamente em termos de eficiência, pois os acessos à memória são mais lineares, reduzindo o número de cache misses e aumentando a taxa de reutilização dos dados na cache.

2.3. Multiplicação de Matrizes por Blocos

Esta abordagem introduz uma otimização adicional dividindo as matrizes em submatrizes menores (blocos) e realizando os cálculos bloco a bloco. Esta técnica permite um melhor aproveitamento, garantindo que os dados acedidos recentemente permanecem armazenados em memória rápida, minimizando assim acessos a níveis de memória mais lentos. Esta implementação melhora substancialmente a eficiência em comparação com os métodos anteriores, no entanto, a sua complexidade mantém-se $O(N^3)$, uma vez que o número de operações não é reduzido, mas sim distribuído de forma mais eficiente.

2.4. Implementações Paralelas

Além das versões sequenciais, desenvolvemos versões paralelas utilizando OpenMP, permitindo explorar a capacidade *multicore* do CPU. As duas principais estratégias de paralelização testadas foram:

- **Paralelização do loop exterior**: A divisão do trabalho é feita uniformemente entre os *cores* disponíveis, ao nível do loop mais externo, criando threads para diferentes porções das linhas da matriz. Esta abordagem melhora substancialmente o desempenho quando comparada com a versão sequencial, uma vez que permite que várias iterações do loop mais externo sejam executadas simultaneamente por diferentes threads.
- **Paralelização do loop interior**: Neste caso, a paralelização ocorre no loop mais interno, onde cada thread é responsável por calcular um subconjunto dos elementos dentro deste, resultando num maior número de operações concorrentes. No entanto, esta abordagem pode ser menos eficiente devido à sincronização necessária entre threads no nível mais profundo da iteração, levando a um overhead adicional.

2.5. Multiplicação de Matrizes Não Quadradas

Para cada método, implementámos a multiplicação de matrizes não quadradas. Para isso, em vez de assumirmos que as matrizes têm dimensões ***N*x*N***, permitimos que tenham diferentes dimensões, ou seja **A(m*x*p)** e **B(p*x*n)** resultando numa matriz **C(m*x*n)**. Ao considerarmos três dimensões diferentes (***m***, ***p*** e ***n***), permitiu-nos uma maior flexibilidade nas simulações.

3. Métricas de Performance

Para termos de comparação dos vários algoritmos desenvolvidos, utilizamos diversas métricas que nos permitiram compreender o impacto das diferentes abordagens e otimizações.

A recolha destas métricas foi realizada através da **Performance API (PAPI)**, que permite aceder a contadores de hardware diretamente da CPU, fornecendo medições precisas sobre o comportamento do programa.

A experimentação envolveu a execução dos algoritmos para diferentes tamanhos de matrizes, desde 600×600 até 10240×10240. Para garantir a precisão dos resultados, cada teste foi repetido cinco vezes e os valores médios foram utilizados para minimizar flutuações causadas por processos em segundo plano ou variações momentâneas do sistema.

Métricas analisadas:

- O **tempo de execução** foi medido como a principal métrica de avaliação de desempenho, uma vez que representa diretamente a eficiência dos algoritmos implementados.
- A análise das **cache misses em L1 e L2** permite compreender que algoritmos são mais cache-friendly e otimizam melhor o uso da hierarquia de memória.
- A medição do número de operações de ponto flutuante realizadas por segundo (**MFLOPS** – Millions of Floating Point Operations Per Second) permite estimar a capacidade computacional da máquina ao executar os diferentes algoritmos, sendo calculada pela fórmula:

$$FLOPS = \frac{2 \times (tamanho\ da\ matriz)^3}{tempo\ de\ execução}$$

- Nas versões paralelas dos algoritmos, o **speedup** e a **eficiência** da execução permitem avaliar qual das versões paralelas apresentou melhor escalabilidade e aproveitamento dos recursos multicore disponíveis.

$$Speedup = \frac{T_{sequencial}}{T_{paralelo}} \qquad Eficiência = \frac{Speedup}{Número\ de\ Threads}$$

- Com algumas destas métricas foi possível realizar também uma comparação entre as versões C++ e Java, analisando o impacto da escolha da linguagem na performance global dos algoritmos.

4. Resultados e Análise

Nesta secção, analisamos os resultados obtidos para as diferentes implementações da multiplicação de matrizes, considerando tanto as versões sequenciais como paralelizadas. Foram realizadas 5 medições e a seguinte avaliação baseia-se em métricas como o tempo de execução, MFLOPS, *cache misses* e eficiência da paralelização.

4.1 Comparação entre Implementações Sequenciais

Os testes com matrizes variando entre 600x600 e 10240x10240 demonstraram diferenças significativas no desempenho dos três métodos sequenciais. A multiplicação convencional revelou-se menos eficiente devido ao elevado número de *cache misses*. A abordagem por linha apresentou uma melhoria substancial, enquanto a multiplicação por blocos obteve o melhor desempenho ao reduzir os acessos à memória principal.

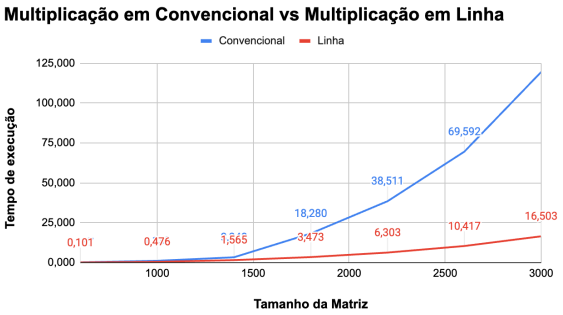

***Figura 1** - Gráfico de comparação entre Multiplicação Convencional e Multiplicação por Linha em C++*

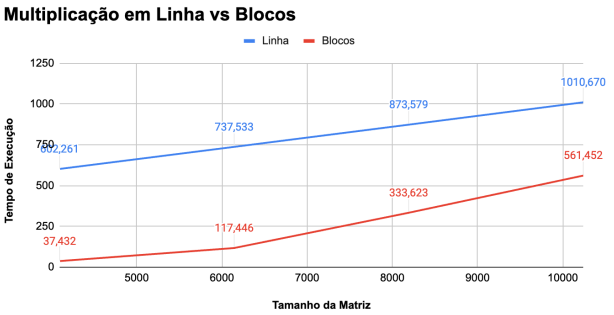

***Figura 2** - Gráfico de comparação entre Multiplicação por Linha e Multiplicação por Blocos em C++*

4.2 Avaliação da Multiplicação por Blocos

Com o objetivo de analisar o efeito do tamanho dos blocos no tempo de execução do algoritmo de multiplicação por blocos, testámos blocos de 128, 256 e 512 em matrizes cujas dimensões variavam de 4096 a 10240, com um incremento de 2048 a cada iteração. Embora tenhamos observado diferenças pequenas no tempo de execução entre os resultados, não conseguimos tirar uma conclusão clara sobre como o tamanho dos blocos influencia o desempenho do algoritmo. Por isso, acreditamos que a análise realizada não permitiu estabelecer uma relação definitiva entre o tamanho dos blocos e o tempo de execução.

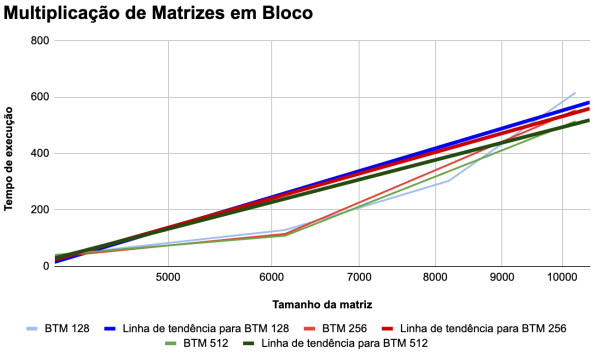

***Figura 3** - Gráfico de comparação entre Multiplicação por Linha e Multiplicação por Blocos em C++*

4.3 Comparação entre as Implementações em C/C++ e Java

Com o objetivo de comparar o desempenho do mesmo algoritmo em diferentes linguagens de programação, executámos o código em C/C++ e em Java para matrizes com tamanhos progressivamente maiores. Após analisarmos os gráficos subsequentes, concluímos que o programa desenvolvido em C/C++ apresentou um desempenho superior ao da versão em Java, com tempos de execução a melhorar à medida que a velocidade do algoritmo aumentava. (Vale relembrar que o Algoritmo de Multiplicação Convencional apresenta um desempenho inferior ao do Algoritmo de Multiplicação por Linha).

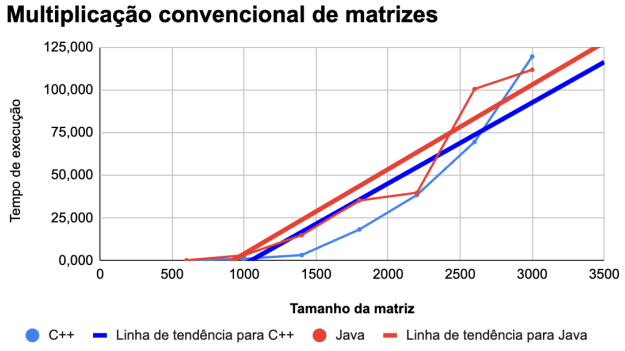

***Figura 4** - Gráfico de comparação entre Multiplicação Convencional em C++ e em Java*

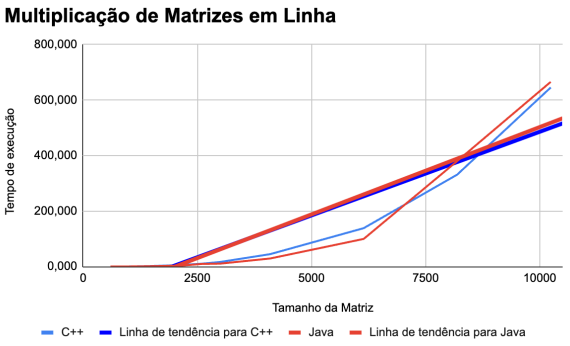

***Figura 5** - Gráfico de comparação entre Multiplicação em Linha em C++ e em Java*

4.4 Comparação entre Algoritmos e Desempenho de Cache

Com o intuito de avaliar o impacto das diferentes abordagens e otimizações no uso da cache, todos os algoritmos foram analisados para matrizes com tamanhos progressivamente maiores. Após a análise dos gráficos seguintes, concluímos que, ao compararmos os algoritmos de Multiplicação Convencional e Multiplicação por Linha, o número de cache misses foi inferior no segundo algoritmo em ambos os níveis de cache. Por outro lado, ao compararmos os algoritmos de Multiplicação por Linha e Multiplicação por Blocos, observou-se uma redução no número de cache misses de nível 1, mas com o custo de um aumento no número de cache misses de nível 2.

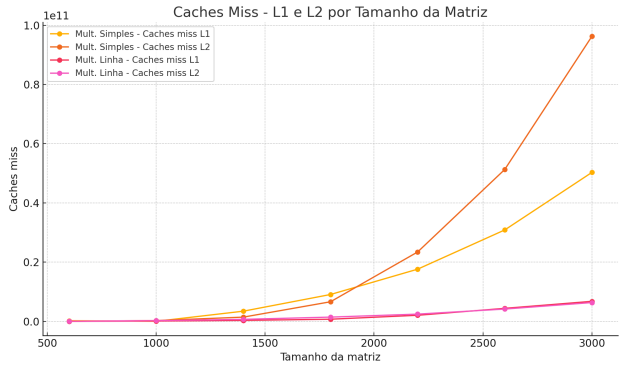

Figura 6 *- Gráfico de comparação de cache misses entre Multiplicação Convencional e Multiplicação em Linha*

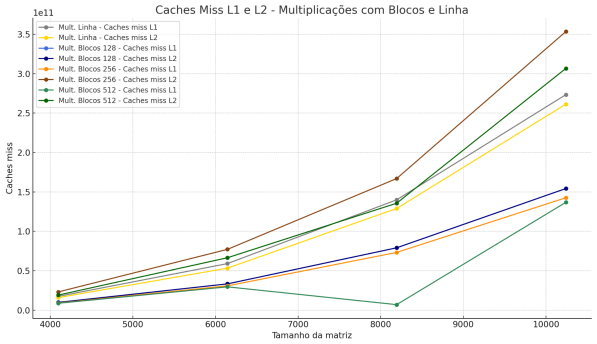

Figura 7 *- Gráfico de comparação de cache misses entre Multiplicação em Linha e Multiplicação em Blocos*

4.5 Comparação entre Implementação Sequencial e Paralela - Multiplicação em Linha

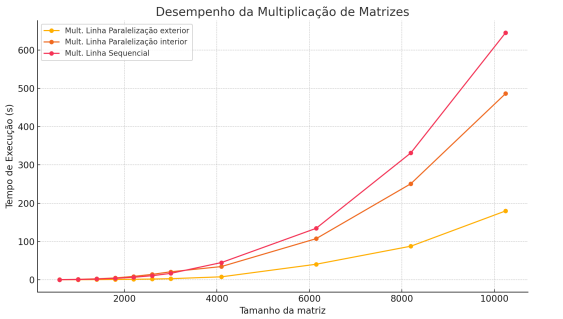

Figura 8 *- Gráfico de comparação do desempenho entre Multiplicação em Linha Sequencial e Paralela Exterior e Interior*

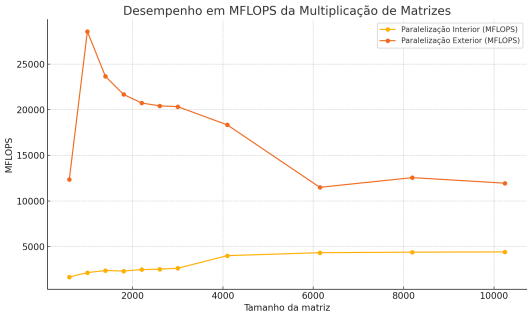

Figura 9 *- Gráfico de comparação do desempenho em MFLOPS entre Multiplicação Paralela Exterior e Interior*

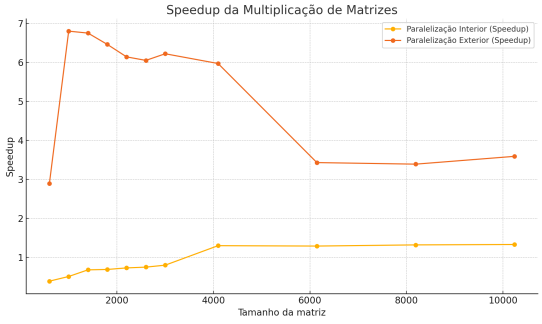

Figura 10 *- Gráfico de comparação do Speedup entre Multiplicação Paralela Exterior e Interior*

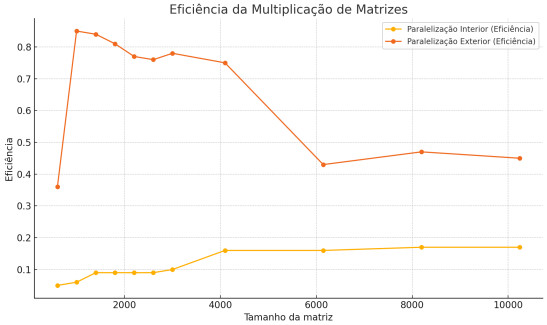

Figura 11 *- Gráfico de comparação de eficiência entre Multiplicação Paralela Exterior e Interior*

A análise do desempenho das versões sequencial e paralela, bem como a comparação entre os dois pragmas utilizados, permite retirar algumas conclusões:

- Ambas as versões paralelizadas apresentam um desempenho superior ao algoritmo sequencial, demonstrando os benefícios do paralelismo na otimização da execução. No entanto, a paralelização exterior mostrou-se mais eficiente do que a paralelização interior. Esta diferença é visível nos gráficos de *SpeedUp*, que ilustram a relação entre o tempo sequencial e paralelo, bem como nos gráficos de MFLOPS, que indicam um maior número

de operações de ponto flutuante por segundo nas versões paralelas.

- A primeira abordagem revela uma eficiência consideravelmente superior à segunda. Como resultado, os núcleos do processador são utilizados de forma mais eficiente, minimizando a sobrecarga associada à sincronização entre *threads* e maximizando o desempenho do algoritmo.

4.6 Tempo de execução da multiplicação de matrizes não quadradas

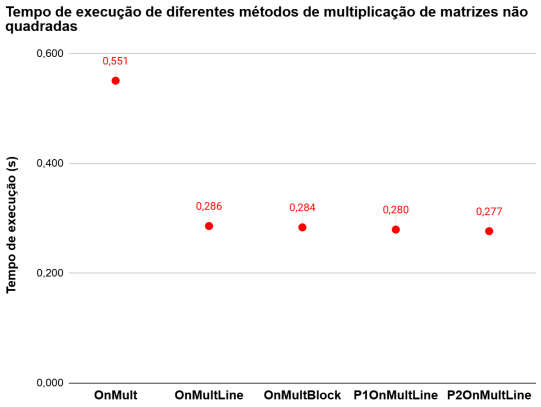

***Figura 11** - Gráfico de comparação entre o tempo de execução dos métodos de Multiplicação single-core e multi-core*

A análise dos tempos de execução confirma as tendências previamente discutidas, demonstrando que a versão multi-core apresenta um desempenho superior à versão *single-core*. Esta melhoria é consistente para diferentes formatos de matriz, evidenciando a eficácia da paralelização na redução do tempo de cálculo. Os gráficos reforçam que a distribuição do processamento por múltiplos *cores* permite um melhor aproveitamento dos recursos do sistema, resultando numa aceleração expressiva da multiplicação de matrizes.

5. Conclusão

Neste projeto, analisamos o impacto da hierarquia de memória e do paralelismo no desempenho da multiplicação de matrizes. Observamos que a multiplicação por linha melhora o uso da cache em relação à abordagem mais simples, enquanto a multiplicação por blocos oferece ganhos ainda maiores reduzindo acessos à memória principal em matrizes de grandes dimensões.

As versões paralelas mostraram que a paralelização do loop exterior é mais eficiente do que a do loop interior, devido à menor sobrecarga de sincronização. A comparação entre C++ e Java demonstrou que C++ apresenta melhor desempenho devido à otimização da gestão de memória.

Este estudo reforça a importância de escolher algoritmos eficientes e considerar a hierarquia de memória para otimizar o desempenho. Pequenos ajustes estruturais e a utilização criteriosa do paralelismo podem resultar em ganhos significativos na execução de operações computacionalmente intensivas.

Anexos

A.1. Multiplicação de Matrizes Simples

A.1.1 - Algoritmo

```C/C++
for(i=0; i<m_ar; i++) {
	for(j=0; j<m_cr; j++) {
		temp = 0;
		for(k=0; k<m_br; k++) {
			temp += pha[i*m_br+k] * phb[k*m_cr+j];
		}
		phc[i*m_cr+j]=temp;
	}
}
```

A.1.2. Tempo de execução em Java

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	0,276	0,26	0,244	0,276	0,247	0,261	0,015
1000	3,246	3,277	3,281	3,12	3,117	3,208	0,083
1400	14,622	14,972	14,886	15,048	15,544	15,014	0,337
1800	35,724	35,701	34,594	34,501	35,85	35,274	0,666
2200	38,426	47,395	44,557	36,025	32,698	39,82	6,06
2600	126,297	77,45	125,983	91,913	80,902	100,509	24
3000	102,067	134,3	100,995	108,689	112,955	111,801	13,498

A.1.3. Tempo de execução em C++

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	0,185	0,187	0,187	0,188	0,187	0,187	0,001
1000	1,027	1,178	1,151	1,135	1,16	1,13	0,06
1400	3,344	3,269	3,332	3,302	3,348	3,319	0,033
1800	18,006	18,315	18,087	18,207	18,787	18,28	0,307
2200	38,008	38,444	38,732	38,652	38,717	38,511	0,304
2600	69,898	69,726	69,778	69,201	69,359	69,592	0,297
3000	119,741	119,859	119,352	119,478	119,254	119,5368	0,256

A.1.4. Caches Miss L1

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	244825356	244822776	244803949	244828261	244839234	244823915	12802
1000	1228213346	1211319689	1211255414	1211266789	1211316436	39891506	885059
1400	3434832536	3457505468	3457545916	3416469224	3517011545	3456672938	37866689
1800	9085223953	9084863199	9084879597	9089372018	9088978415	9086663436	2301662
2200	17630727166	17632791561	17630878999	17630701517	17652498045	17635519458	9531872
2600	30882470504	30907665652	30877059038	30875875667	30877805058	30884175184	13367947
3000	50305738812	50295663277	50293086457	50306098359	50293855947	50298888570	6486548

A.1.5. Caches Miss L2

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	39833885	40137284	39027426	39206111	41252826	39891506	885059
1000	321372850	264879312	306966170	289513723	293624968	295271405	21080761
1400	1444721039	1656942351	1263138418	1525311345	1359602780	1449943187	151338771
1800	8036694992	5361773939	6725125593	5225079396	7811841553	6632103095	1319731627
2200	23510159297	21824755591	23890271355	24093847310	23876857458	23439178202	926692841
2600	51818065021	51418227869	51303912524	51297725812	50660592969	51299704839	415701392
3000	97160733464	95731300530	97042474807	94879630582	96363474123	96235522701	950820916

A.2. Multiplicação de Matrizes por Linha

A.2.1. Algoritmo

```
C/C++
for(i=0; i<m_ar; i++){
	for(k=0; k<m_br; k++){
		for(j=0; j<m_cr; j++){
			phc[i*m_cr+j] += pha[i*m_br+k] * phb[k*m_cr+j];
		}
	}
}
```

A.2.2. Tempo de execução em Java

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	0,161	0,247	0,12	0,119	0,12	0,153	0,055
1000	0,413	0,445	0,395	0,423	0,429	0,421	0,019
1400	1,148	1,102	1,112	1,101	1,091	1,111	0,022
1800	2,292	2,254	2,306	2,296	2,307	2,291	0,022
2200	4,133	4,167	4,117	4,13	4,122	4,134	0,02
2600	11,181	10,95	10,524	8,01	6,79	9,491	1,971
3000	10,441	10,359	10,346	11,315	10,291	10,55	0,431
4096	26,805	30,529	29,959	28,338	29,934	29,113	1,527
6144	100,794	98,618	99,428	99,326	100,214	99,676	0,843
8192	310,783	399,559	400,715	379,767	410,19	380,203	40,355
10240	505,557	709,709	711,871	703,344	698,354	665,767	89,719

A.2.3. Tempo de execução em C++

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	0,1	0,104	0,101	0,101	0,101	0,101	0,002
1000	0,475	0,473	0,475	0,478	0,48	0,476	0,003
1400	1,572	1,575	1,553	1,55	1,576	1,565	0,013
1800	3,37	3,413	3,405	3,356	3,82	3,473	0,196
2200	6,312	6,286	6,305	6,321	6,293	6,303	0,014
2600	10,379	10,359	10,414	10,544	10,39	10,417	0,074
3000	15,968	16,042	16,383	16,803	17,321	16,503	0,564
4096	41,092	42,84	49,538	49,252	41,041	44,753	4,301
6144	139,035	139,15	138,174	137,867	137,852	138,416	0,632
8192	326,065	332,495	331,083	335,559	332,021	331,445	3,444
10240	644,314	648,562	643,898	642,504	648,379	645,531	2,766

A.2.4. Caches Miss L1

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	27152808	27155388	27155718	27156417	27155713	27155209	1394
1000	125823898	125835673	125839840	125834655	125833517	125833517	5883
1400	346256651	346067051	346065554	346069610	346068673	346105508	84506
1800	745605458	744702812	744702703	744689991	744689479	744878089	406664
2200	2074327832	2074070672	2074728264	2074647839	2074573583	2074469638	268668
2600	4413142062	4413160190	4413189957	4413213705	4413168105	4413174804	27719
3000	6780788375	6781056943	6780766913	6780677746	6780689588	6780795913	153544
4096	17541272134	17527854151	17529834539	17531516207	17529248336	17531945073	5376811
6144	59079529001	59160862147	59078844448	59084461700	59084461752	59097631810	35445925
8192	140045444316	139943133965	140177308623	139955010791	140180832057	140060345950	115383461
10240	273187631256	273617842525	273643443511	273218140987	273605250172	273454461690	230320726

A.2.5. Caches Miss L2

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	56848711	58292064	58218750	58192335	58294203	57969212,6	627979
1000	262240303	261728440	262098197	262127281	262084448	262055733,8	192960
1400	701466356	701541987	704130227	696317271	702839256	701259019,4	2970055
1800	1445735675	1462379796	1454181013	1460106981	1456537818	1455788257	6447853
2200	2550101289	2542583009	2580323499	2074647839	2588136606	2467158448	220269722
2600	4206390454	4208178558	4149567728	4166144026	4147143782	4175484910	29942442
3000	6402686594	6263928221	6416940146	6379649760	6381882884	6369017521	60729950
4096	16284963230	15864232367	15865028794	15871551055	15851657155	15947486520	188792560
6144	52905852276	54930880509	52963909190	52870345131	52870344158	53308266253	907874784
8192	129330981218	126725407300	130612044102	127327610134	130978610619	128994930675	1910082856
10240	261067645194	270429561314	261561721048	252507175617	261493548179	261411930270	6339557478

A.3. Multiplicação de Matrizes por Linha - Paralelização do Loop Exterior

A.3.1. Algoritmo

```C/C++
# pragma omp parallel for
for (int i = 0; i < m_ar ; i ++) {
	for (int k = 0; k < m_br ; k ++) {
		for (int j = 0; j < m_cr ; j ++) {
			phc[i*m_cr+j] += pha[i*m_br+k] * phb[k*m_cr+j];
		}
	}
}
```

A.3.2. Tempo de execução em C++

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	0,113	0,014	0,015	0,016	0,015	0,035	0,044
1000	0,07	0,069	0,069	0,069	0,07	0,07	0,001
1400	0,231	0,234	0,233	0,23	0,233	0,232	0,002
1800	0,539	0,533	0,534	0,54	0,542	0,538	0,004
2200	1,006	1,006	1,09	1,017	1,014	1,027	0,036
2600	1,681	1,682	1,67	1,665	1,908	1,721	0,105
3000	2,6054	2,7148	2,6305	2,603	2,7234	2,655	0,059
4096	7,455	7,642	7,548	7,293	7,534	7,494	0,131
6144	42,359	47,867	43,203	33,875	34,522	40,365	6,012
8192	83,6143	82,7339	82,2992	85,1189	104,115	87,576	9,308
10240	161,434	184,554	165,704	184,387	202,505	179,717	16,546

A.3.3. Caches Miss L1

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	3439962	3437280	3439849	3441535	3440169	3439759	1542
1000	15841521	15843787	15844222	15841479	15842952	15842792	1265
1400	43704235	43670655	43670185	43672759	43672208	43678008	14700
1800	93979127	93896587	93902823	93912525	93916392	93921491	33155
2200	259233497	259153973	259157067	259224726	259210647	259195982	37841
2600	550465431	550479028	550493569	550482249	550297314	550443518	82344
3000	846158899	846167153	846166327	846165727	846170743	846165770	4306
4096	2201570899	2201299765	2201690536	2202496627	2202277528	846872269	1114958507
6144	7412315620	7410401691	7414859658	7422735156	7426952789	7417452983	7088480
8192	17593357784	17600984581	17586805724	17591286382	17557361917	17585959278	16788398
10240	34372426201	34333742005	34366355450	34329417338	34314399392	34343268077	24996065

A.3.4. Métricas

Dimensões	Média Tempo S.	Média Tempo P.	Threads	MFLOPS	Speedup	Eficiência
600	0,101	0,035	8	12342,86	2,886	0,361
1000	0,476	0,07	8	28571,43	6,800	0,850
1400	1,565	0,232	8	23655,17	6,746	0,843
1800	3,473	0,538	8	21680,30	6,455	0,807
2200	6,303	1,027	8	20736,12	6,137	0,767
2600	10,417	1,721	8	20425,33	6,053	0,757
3000	16,503	2,655	8	20338,98	6,216	0,777
4096	44,753	7,494	8	18339,87	5,972	0,746
6144	138,416	40,365	8	11491,55	3,429	0,429
8192	331,445	87,576	8	12554,94	3,785	0,473
10240	645,531	179,717	8	11949,25	3,592	0,449

A.4. Multiplicação de Matrizes por Linha - Paralelização do Loop Interior

A.4.1. Algoritmo

```
C/C++
# pragma omp parallel
for (int i = 0; i < m_ar ; i ++) {
       for (int k = 0; k < m_br ; k ++) {
              # pragma omp for
              for (int j = 0; j < m_cr ; j ++) {
                     phc[i*m_cr+j] += pha[i*m_br+k] * phb[k*m_cr+j];
              }
       }
}
```

A.4.2. Tempo de execução em C++

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	0,251	0,288	0,252	0,253	0,25	0,259	0,016
1000	0,924	0,86	0,93	0,881	1,071	0,933	0,082
1400	2,27	2,128	2,772	2,172	2,209	2,31	0,263
1800	4,386	4,974	5,298	4,879	5,652	5,038	0,474
2200	8,451	8,646	8,716	8,648	8,586	8,609	0,1
2600	14,468	13,812	13,883	13,721	13,825	13,942	0,3
3000	20,53	21,152	20,325	20,171	20,96	20,627	0,417
4096	33,659	33,6365	33,4416	35,2648	35,6828	34,337	1,052
6144	109,536	106,487	106,455	106,459	107,168	107,221	1,329
8192	250,07	249,983	252,254	251,885	249,983	250,835	1,135
10240	489,349	481,972	487,072	484,122	489,944	486,492	3,408

A.4.3. Caches Miss L1

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
600	7983078	7973059	7962906	7985906	7955361	7972062	13007
1000	28764476	28714337	28797682	28724450	28692255	28738640	42122
1400	68518873	67193910	67123940	67123940	68279046	67647942	691396
1800	133711378	132744539	132759236	132775772	132848094	132967804	417569
2200	231819166	231678096	231689338	231801052	231504115	231698353	125860
2600	363804232	362827857	362827857	363577712	363749531	363357438	490610
3000	538867011	540023701	539854429	539283835	540136210	539633037	539341
4096	1231706933	1229915014	1226308726	1229266338	1230194093	1229478221	1985007
6144	4053621439	4065003218	4070851891	4058906320	4064385245	4062553623	6544145
8192	9336103624	9314105125	9321731454	9312012695	9314105125	9319611605	9935624
10240	18266792936	18264341697	18271512240	18269596322	18291553584	18272759356	10854011

A.4.4. Métricas

Dimensões	Média Tempo S.	Média Tempo P.	Threads	MFLOPS	Speedup	Eficiência
600	0,101	0,259	8	1667.95	390	49
1000	0,476	0,933	8	2143.62	510	64
1400	1,565	2,31	8	2375.76	677	85
1800	3,473	5,038	8	2315.20	689	86
2200	6,303	8,609	8	2473.69	732	92
2600	10,417	13,942	8	2521.30	747	93
3000	16,503	20,627	8	2617.93	800	100
4096	44,753	34,337	8	4002.65	1.303	163
6144	138,416	107,221	8	4326.17	1.291	161
8192	331,445	250,835	8	4383.41	1.321	165
10240	645,531	486,492	8	4414.22	1.327	166

A.5. Multiplicação de Matrizes por Blocos

A.5.1. Algoritmo

```C/C++
for(bi=0; bi<m_ar; bi+=bkSize){
    for(bj=0; bj<m_cr; bj+=bkSize){
        for(bk=0; bk<m_br; bk+=bkSize){
            for(i=bi; i<min(bkSize+bi, m_ar); i++){
                for(k=bk; k<min(bkSize+bk, m_br); k++){
                    for(j=bj; j<min(bkSize+bj, m_cr); j++){
                        phc[i*m_cr+j] += pha[i*m_br+k] * phb[k*m_cr+j];
                    }
                }
            }
        }
    }
}
```

A.5.2. Tempo de execução em C++

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
Blocos de 128							
4096	37,382	37,365	38,12	39,13	38,071	38,014	0,721
6144	129,041	129,76	129,363	125,9	129,754	128,764	1,629
8192	318,67	297,583	283,049	313,369	302,365	303,007	13,972
10240	611,731	623,788	622,46	610,191	612,393	616,113	6,467
Blocos de 256							
4096	37,97	32,52	32,94	32,729	33,198	33,871	2,305
6144	114,841	115,559	116,119	112,466	115,787	114,954	1,468
8192	341,386	371,958	325,686	375,71	390,398	361,028	26,611
10240	556,528	552,843	554,576	552,808	551,722	553,695	1,885
Blocos de 512							
4096	40,993	40,107	40,491	40,831	39,631	40,411	0,553
6144	108,912	108,777	108,031	107,469	109,91	108,62	0,928
8192	339,496	334,06	337,128	337,48	336,012	336,835	1,998
10240	514,97	518,439	516,106	514,301	508,934	514,55	3,512

A.5.3. Caches Miss L1

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
Blocos de 128							
4096	9891634791	9891719570	9892298900	9891866107	9891531084	9891810090	299517
6144	33347980801	33360535251	33349809920	33367769640	33358693218	33356957766	8129951
8192	79132148245	79113499734	79127805969	79147219904	79120635766	79128261924	12755361
10240	154340475910	154289846805	154345909198	154341738261	154328855174	154329365070	22979382
Blocos de 256							
4096	9135964021	9135964021	9136669669	9135764187	9137271478	9136326675	630422
6144	30827107336	30829288487	30829477748	30829643561	30830542578	30829211942	1271098
8192	73186561243	73299415715	73153563588	73260137209	73367008708	73253337293	85884346
10240	142638656874	142697269058	142647405990	142646597060	142677390909	142661463978	24872109
Blocos de 512							
4096	8760172904	8760580281	8758351748	8760078250	8760213469	8759879330	874963
6144	29602844831	29601533275	29613457763	29598837593	29606111840	29604557060	5621992
8192	70229043128	70199390153	70222239076	70226486354	70226189997	70220669742	12141906
10240	136859186649	136836671695	136865212769	136851059768	136891139691	136860654114	20121587

A.5.4. Caches Miss L2

Dimensões	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5	Média	D. Padrão
Blocos de 128							
4096	33178337141	33180468620	33156334817	33129073353	33498282153	33228499217	152232723
6144	110973228809	110995830790	111196194966	111607548203	110963468138	111147254181	274434405
8192	263691840342	268186604112	263273658028	261627476851	267787015510	264913318969	2913322154
10240	516993084838	518153588731	510767892654	512226710438	512689604776	514166176287	3216402991
Blocos de 256							
4096	23114316916	23114316916	23336959037	23084038379	23085189947	23146964239	107244400
6144	77187232539	77193260260	77112524656	77537475974	76688468388	77143792363	303157756
8192	172273585218	162189541297	175636562821	164982960491	159894591011	166995448168	6710003969
10240	354405353372	355446996321	353031938635	355088631470	349591800468	353512944053	2378364808
Blocos de 512							
4096	19125263881	19113951658	18880164770	19323166369	19265335148	19141576365	171505400
6144	66759535985	66724425856	66284421801	66222854033	66606773716	66519602278	250244440
8192	134745177450	136857388329	135069872630	134851767629	136968009115	135698443031	1115305731
10240	306895815987	304680609571	310458230534	304305678833	307277525417	306723572068	2464959918

A.6. Multiplicação de Matrizes Não Quadradas

A.6.1. Métricas para multiplicação das matrizes A(600*1200) e B(1200*800)

Nota: na multiplicação por blocos foram utilizados bloco de 128.

Dimensões	Tempo de execução	Caches miss L1	Caches Miss L2
Multiplicação de Matrizes Simples	0,551	605788616	602959195
Multiplicação de Matrizes por Linha	0,2862	72467671	155993636
Multiplicação de Matrizes por Blocos	0,2838	85574104	87243368
Multiplicação de Matrizes por Linha - Paralelização Loop Exterior	0,2798074	72507285	157021804
Multiplicação de Matrizes por Linha - Paralelização Loop Interior	0,2768188	72501975	157126736