FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório de Computação Paralela e Distribuída - Projeto 1

2.º Semestre

2024/2025

Turma 12 - Grupo 12
Lucas Faria - up202207540
Pedro Borges - up202207552
Alexandre Lopes - up202207015

Índice

1.Descrição do Problema

Neste projeto, analisaram-se as diferenças de desempenho entre diversos métodos de implementação do algoritmo de multiplicação de matrizes. Adicionalmente, comparou-se o desempenho de duas linguagens de programação distintas para a resolução do mesmo problema: C++ e Python. Durante o desenvolvimento do trabalho, também foram exploradas as variações de performance entre soluções de processamento *single-core* e *multi-core*, avaliando como diferentes abordagens de implementação impactam os resultados obtidos.

2.Algoritmos

Os algoritmos desenvolvidos neste projeto podem ser categorizados em duas grandes classes: *single-core* e *multi-core*. Os algoritmos *single-core* utilizam computação sequencial para resolver o problema da multiplicação de matrizes, diferenciando-se principalmente na eficiência no uso da memória, que pode ser mais ou menos otimizada. Por outro lado, os algoritmos *multi-core* recorrem à computação paralela, distinguindo-se sobretudo pela estratégia adotada na distribuição do trabalho entre *threads* e na sincronização das mesmas.

2.1 Multiplicação simples de matrizes

Este é o algoritmo mais simples para realizar a multiplicação de matrizes. Dadas duas matrizes, A e B, a multiplicação entre elas consiste em calcular o produto escalar de cada linha de A com cada coluna de B. Este algoritmo apresenta uma complexidade de O(n^3). No anexo A1.1, apresenta-se o código correspondente em C++.

2.2 Multiplicação de matrizes por linha

Este algoritmo representa uma evolução em relação à multiplicação simples de matrizes. Neste caso, ao considerar duas matrizes A e B, cada elemento da matriz A é multiplicado pela linha correspondente da matriz B, permitindo uma utilização mais eficiente da memória. Como o acesso é realizado linha a linha em B, são necessárias menos atualizações na cache, uma vez que os blocos de memória correspondentes às linhas de B são contíguos em memória. Assim como no método de multiplicação simples, a complexidade computacional deste algoritmo permanece O(n^3). O código em C++ correspondente a esta implementação é apresentado no anexo A1.2.

2.3 Multiplicação de matrizes por bloco

Este algoritmo é uma variação do método de multiplicação por linha, no qual as matrizes são divididas em blocos menores para realizar os cálculos. A principal vantagem desta abordagem é a melhoria significativa no uso da cache, já que apenas pequenos blocos das matrizes são carregados na cache de cada vez, reduzindo assim o número de acessos à memória principal. Isso resulta numa maior eficiência no acesso à memória e no desempenho geral.

Apesar da melhoria na gestão da cache, a complexidade do algoritmo permanece O(n^3). No anexo A1.3, apresenta-se o código correspondente em C++ para esta implementação.

2.4 Multiplicação de matrizes usando computação paralela - versão 1

Este algoritmo utiliza computação paralela para resolver o problema de multiplicação de matrizes. Nesta implementação, o ciclo *for* mais externo é paralelizado, permitindo que os *threads* do CPU executem partes distintas do ciclo simultaneamente. Dessa forma, o trabalho é eficientemente distribuído entre os *threads*, aproveitando os recursos de múltiplos núcleos do processador. Embora a paralelização melhore a performance, a complexidade temporal do algoritmo ainda permanece O(n^3), como no método de multiplicação por linha. No anexo A1.4, apresenta-se o código correspondente em C++ para esta implementação.

2.5 Multiplicação de matrizes usando computação paralela - versão 2

Este algoritmo distingue-se da versão 1 pelo fato de o ciclo *for* paralelizado ser o mais interno. Assim como na versão anterior, a complexidade temporal deste algoritmo permanece O(n^3). O código correspondente em C++ para esta implementação é apresentado no anexo A1.5.

3.Métricas de Performance

Para avaliar o desempenho dos diferentes algoritmos, utilizamos a ferramenta **PAPI**. Com esta ferramenta, foi possível recolher as métricas de *cache misses* nas caches L1 e L2. Estas medições são cruciais, pois os *cache misses* podem impactar significativamente o desempenho, uma vez que geram acessos à memória principal, que são mais lentos.

Para além disso, medimos o tempo de execução de cada algoritmo, tanto nas versões implementadas em C++ como nas desenvolvidas em Python. Em conjunto com os tempos de execução, também calculamos o número de MFLOPS (*Mega Floating Point Operations Per Second*) alcançado por cada algoritmo.

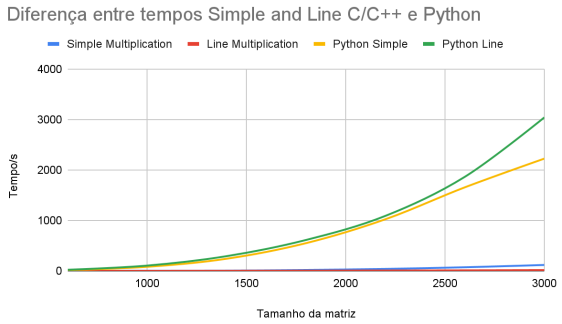
Para garantir a confiabilidade dos resultados, cada algoritmo foi executado entre duas a três vezes, minimizando interferências de outros processos em execução no computador. Além disso, utilizamos a flag de otimização de desempenho -O2 no compilador de C++, de forma a obter um código mais eficiente. Todos os resultados foram obtidos através do mesmo computador (computador da Feup), à exceção dos algoritmos em python, visto que estes demoraram demasiado no dado computador.

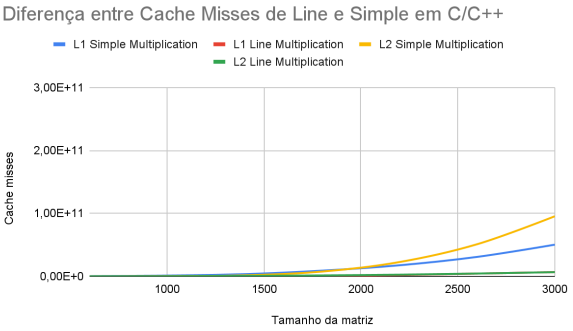
4.Análise dos resultados

4.1 Multiplicação simples vs. multiplicação por linha

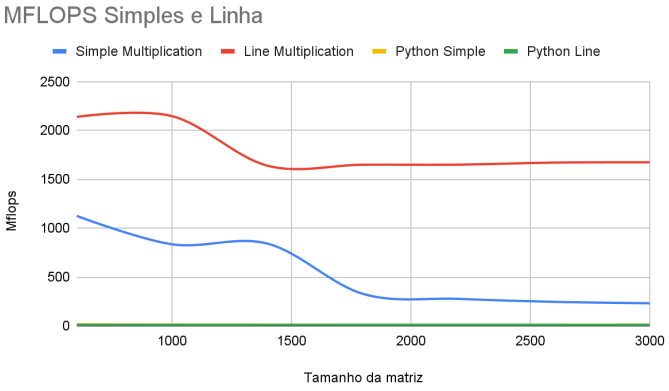
Python, sendo uma linguagem interpretada, apresenta sobrecarga adicional na execução, tornando-a significativamente mais lenta do que C++, que é compilada e otimizada para desempenho. Para além disso, enquanto Python aloca memória automaticamente, C++ permite uma alocação mais eficiente da mesma. Neste caso, não foi possível recorrer a bibliotecas externas, como NumPy, para otimizar o desempenho.

Em relação aos algoritmos, a multiplicação por linha foi mais eficiente que a multiplicação simples. Observou-se uma redução significativa nas *cache misses* (as linhas vermelha e verde estão praticamente sobrepostas), o que se refletiu num tempo de execução menor. Ao utilizar a matriz resultante como acumulador e um ciclo "for" para aceder linha a linha, garante-se a disponibilidade dos dados nos níveis inferiores de memória.





Devido à menor eficiência do Python, os Mflops calculados para os algoritmos executados nessa linguagem foram muito baixos, atingindo um máximo de 12 MFLOPS. Por outro lado, a versão em C++ apresentou melhores resultados, visto que a versão por linha obteve um maior número de MFLOPS. Contudo, ambos os algoritmos apresentaram uma redução no número de MFLOPS à medida que o tamanho da matriz aumentava, devido ao maior número de cache misses, impactando negativamente o acesso à memória. Este padrão foi claramente identificado no gráfico abaixo.



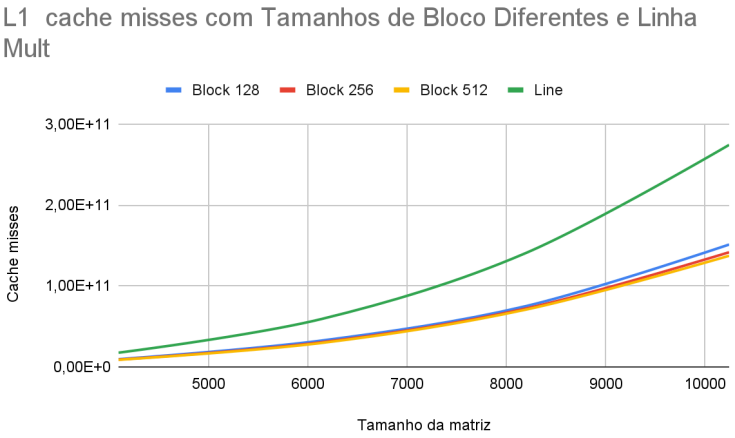
4.2 Multiplicação por linha vs. multiplicação por blocos

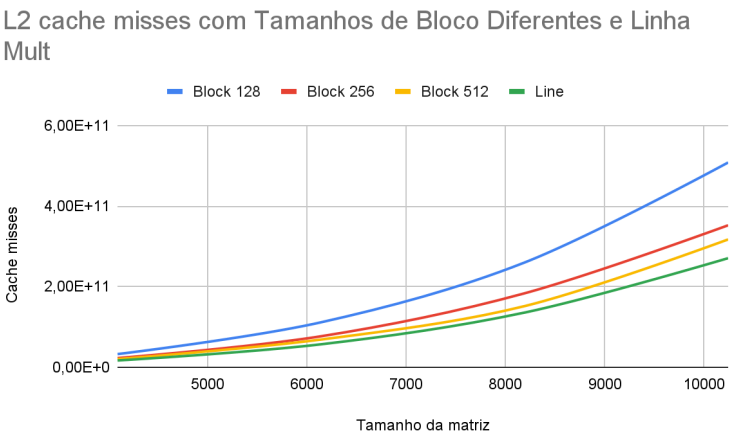
Como é possível observar nos gráficos abaixo, o número de *cache misses* de L1 foi significativamente inferior na multiplicação por blocos, enquanto na cache L2 as diferenças foram menos acentuadas, especialmente entre o bloco de 512 e a multiplicação por linha. Apesar de as *cache misses* de L2 serem mais numerosas na multiplicação por blocos, as *cache misses* de L1 são mais dispendiosas, exigindo mais tempo para recuperar os dados.

A diferença no tamanho dos blocos não teve um grande impacto nas *cache misses* de L1. Uma possível explicação para este comportamento é que a cache L1 tem 128 KB, o que corresponde ao tamanho do bloco de 128. Assim, não há diferença notável ao usar blocos maiores. No entanto, para blocos de 256 e 512, que ocupam, 512 KB e 2 MB, respetivamente, a

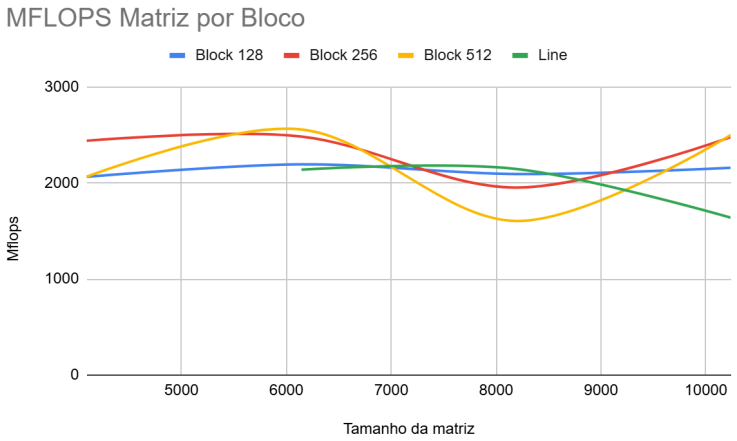
redução de *misses* na cache L2 é mais evidente devido ao maior aproveitamento do espaço disponível nessa cache.

Além disso, o bloco de 512 não só apresenta menos *cache misses* como também menor tempo de execução.





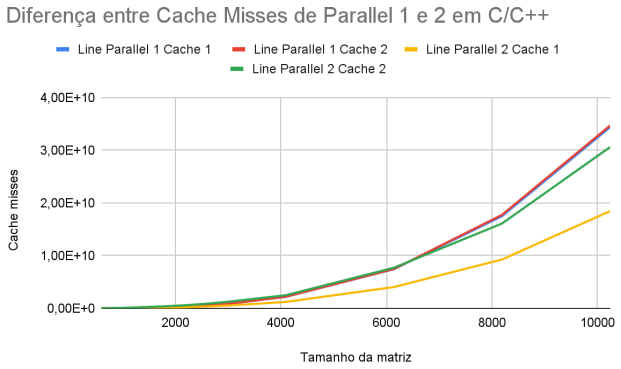
Por estas razões, os MFLOPS da multiplicação por blocos são superiores aos da multiplicação por linha, conforme esperado.

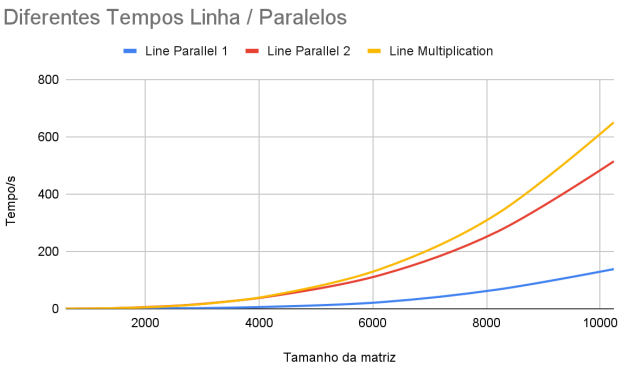


4.3 Multiplicação usando computação paralela - versão 1 vs. versão 2

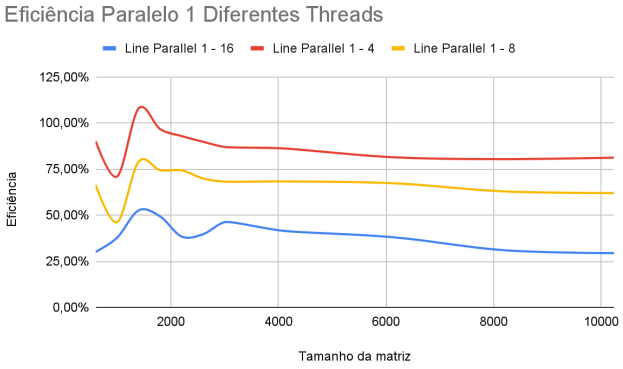
A principal diferença entre as duas versões reside na escolha do ciclo paralelizado. Na versão 1, o ciclo mais externo foi paralelizado, distribuindo eficientemente a carga de trabalho entre os threads, ainda que com maior concorrência no acesso à cache. Já na versão 2, a paralelização ocorreu no ciclo mais interno, permitindo que cada thread processasse blocos menores de dados, reduzindo a interferência entre threads e reduzindo a frequência de acessos à memória principal ao aproveitar melhor os dados já armazenados na cache.

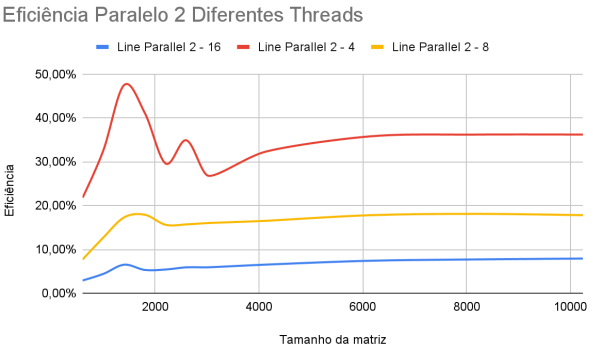
Os resultados demonstram que a versão 2 tem um melhor aproveitamento da memória cache, o que resulta num menor número de *cache misses* e, consequentemente, num tempo de execução reduzido.



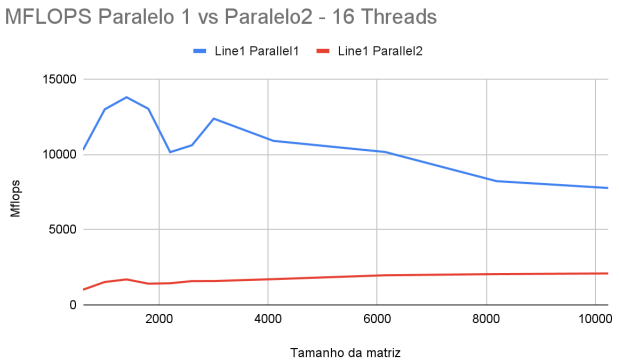


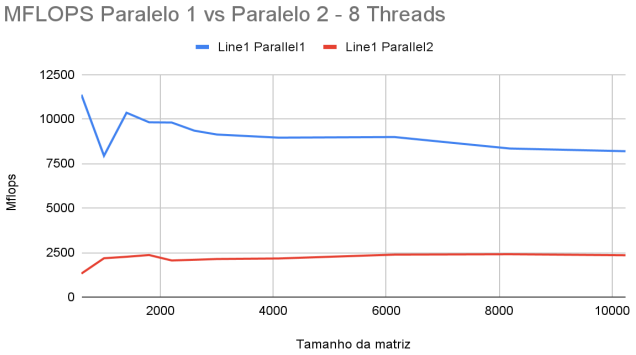
Os melhores tempos obtidos pela versão 1 indicam que esta utiliza os recursos da CPU e da memória de forma mais eficiente. Os gráficos abaixo ilustram a eficiência dos dois algoritmos para diferentes números de threads. É importante notar que, em ambos os casos, um menor número de threads resulta em maior eficiência, sugerindo que, para esses algoritmos, pode fazer mais sentido usar menos threads se quisermos utilizar de forma mais eficiente os recursos da CPU.

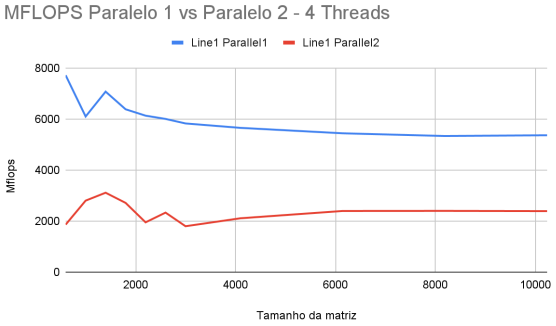




Ao comparar os MFLOPS dos dois algoritmos, observamos que a primeira versão apresenta um desempenho superior, independentemente do número de threads em uso. Por outro lado, na segunda versão, o desempenho piora à medida que mais threads são usados. Isto ocorre porque a versão 2 requer a execução n^2 sincronizações de threads devido ao paralelismo no ciclo “for” mais interno, enquanto que a primeira versão necessita de apenas uma sincronização.





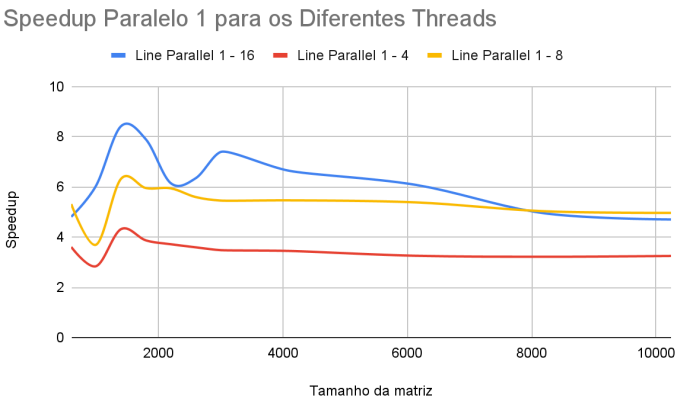


4.4 Multiplicação usando computação paralela - versão 1 e versão 2 vs. multiplicação por linha

Ao comparar as versões paralelas com a multiplicação por linha, verificamos que a versão 1 paralela apresenta uma execução significativamente mais rápida. Esta alcança um *speedup* de até 8.4x ao utilizar 16 threads com uma matriz 1400X1400. Por outro lado, a versão 2 paralela apresenta desempenho inferior ao da multiplicação por linha, sendo que, no pior cenário, a performance é reduzida para metade quando ao utilizar 16 threads com uma matriz de 600X600.

Os resultados obtidos pela versão 1 explicam-se pela necessidade de realizar apenas uma sincronização, permitindo dividir uma quantidade maior de trabalho entre os threads. Assim, os ganhos aumentam proporcionalmente ao número de threads utilizados no algoritmo. Em contraste, na versão 2, como apenas o ciclo “for” mais interno é paralelizado, são necessárias n^2 sincronizações, e apenas o trabalho dentro desse ciclo é distribuído entre os threads. Isso faz com que a versão 2 seja mais penalizada à medida que o número de threads aumenta, tendo em conta que serão mais threads para sincronizar nas n^2 sincronizações.

Os gráficos abaixo ilustram os ganhos de velocidade da versão 1 paralela e da versão 2 paralela em relação à multiplicação por linha.



5.Conclusões

Com a realização deste trabalho, constatamos que pequenas diferenças nos algoritmos podem ter um impacto significativo na performance. Por exemplo, uma simples mudança na forma com acedemos à memória e realizamos a multiplicação das matrizes no algoritmo de multiplicação por linha resultou numa diferença substancial de desempenho quando comparado com o algoritmo simples, apesar de ambos terem a mesma complexidade temporal.

Além disso, verificamos que a forma como dividimos o problema pode impactar significativamente a performance. No algoritmo por bloco, a divisão das matrizes em blocos impactou a performance de forma positiva, especialmente em blocos de tamanhos maiores. Por outro lado, nos algoritmos paralelos, uma paralelização num ciclo mais externo permitiu um maior ganho de performance devido à menor necessidade de sincronização.

Em suma, aprendemos que, mesmo quando diferentes versões de algoritmos apresentam complexidades temporais e espaciais semelhantes, isso não implica que tenham desempenhos equivalentes. Alterações no código podem tornar um algoritmo mais eficiente que outro, ao explorar melhor os recursos de hardware.

A.Anexos

A1.Código dos Algoritmos

A1.1 Multiplicação Simples

```C/C++
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 +j] = pha[i * m_ar + k] * phb[k * m_br + j];
```

```
            }

      }
}
```

A1.2 Multiplicação por Linha

```C/C++
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_br + k];
            }
      }
}
```

A1.3 Multiplicação por Bloco

```C/C++
for(i = 0; i < m_ar; i+=bkSize) {
      for(j = 0; j < m_br; j+=bkSize) {
            for(k = 0; k < m_ar; k+=bkSize) {
                  for (ii = i; ii < min(i + bkSize, m_br) ; ii++){
                        for (jj = j; jj < min(j + bkSize, m_br); jj++){
                              for(kk = k ; kk < min(k + bkSize, m_br); kk++){

                                    phc[ii * m_ar + kk] += pha[ii * m_ar + jj]
* phb[jj * m_br + kk];
                              }
                        }
                  }
            }
      }
}
```

A1.4 Multiplicação usando computação paralela - versão 1

```C/C++
#pragma omp parallel private (j,k)
{
	#pragma omp for
	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_br +
k];
			}
		}
	}

}
```

A1.5 Multiplicação usando computação paralela - versão 2

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

A2. Resultados Recolhidos

A2.1 Tempos e cache misses por tamanho de matriz

Tempos

Matrix Size	Simple Multiplication	Line Multiplication	Block 128	Block 256	Block 512	Line Parallel 1	Line Parallel 2	Python Simple	Python Line
600	0,192	0,101				0,021	0,214	14,565	22,084
1000	1,199	0,466				0,077	0,658	80,802	104,629
1400	3,264	1,675				0,199	1,622	240,557	291,548
1800	17,894	3,536				0,448	4,138	549,366	612,316
2200	38,63	6,454				1,05	7,423	1023,176	1090,153
2600	71,695	10,519				1,658	11,128	1655,645	1864,083
3000	116,966	16,13				2,183	17,07	2223,669	3039,649
4096		41,932	33,294	28,163	33,291	6,31	40,205		
6144		138,834	105,693	93,416	90,687	22,843	117,962		
8192		331,47	262,649	281,545	342,639	66,919	268,989		
10240		650,435	497,445	433,027	429,183	138,331	514,427		

L1 cache misses

Matrix Size	L1 Simple Multiplication	L1 Line Multiplication	Block 128	Block 256	Block 512	Line Parallel 1	Line Parallel 2
600	244734279	27108225				1072588	3734539
1000	1224014337	125684730				5119200	17176097
1400	3489067282	346111126				24576636	40145866
1800	9074707630	747548573				69406425	78619280
2200	17630611531	2078508365				131621222	131720170
2600	30903187637	4413657780				219774106	178885734
3000	50299695289	6780758412				324497115	302300488
4096		17581404327	9526964679	9021903267	8796750853	860986941	693224491
6144		59316983751	32620014643	30553269808	29670096508	308709480 1	219016072 1
8192		140489506764	74977297566	72056482201	70668374568	727380815 5	478363582 6
10240		274177238107	151106628050	141571232834	137224185723	147022962 96	962783439 5

L2 cache misses

Matrix Size	L2 Simple Multiplication	L2 Line Multiplication	Block 128	Block 256	Block 512	Line Parallel 1	Line Parallel 2
600	39042043	58497095				2387540	11356510
1000	288989652	269319897				10786900	53197785
1400	1600615429	717937503				29252365	114653992
1800	7590700887	1489666427				53244186	163009200
2200	22758092932	2668839618				102769473	231478205
2600	51058921134	4484819516				173760465	267044495
3000	95399035999	6671881017				245039868	457055882
4096		16902147947	32704852138	22878473082	20383218764	661151258	12910983532
6144		57046237610	11158880667 0	76971038189	68867334115	25593342688	41365768433
8192		135686964195	25987207 2180	183494457423	151704112015	63313845664	83993446665
10240		270580545337	507858274965	352159824365	317077074116	18863009747	14526972488

Tempos e cache misses 4 threads

Matrix Size	Line Parallel 1	Line Parallel 2	Line Parallel 1 Cache 1	Line Parallel 1 Cache 2	Line Parallel 2 Cache 1	Line Parallel 2 Cache 2
600	0,028	0,116	6780005	14587280	10467138	36674329
1000	0,164	0,357	31434100	68619247	43646885	135585018
1400	0,388	0,882	86919043	184322920	105179096	288893423
1800	0,914	2,15	187149882	378903615	220788220	558060744
2200	1,737	5,459	517508927	685704195	374874000	825624918
2600	2,928	7,539	1099460292	1133849542	625948822	1347192929
3000	4,635	15,016	1690209387	1699534569	926424266	1824521049
4096	12,157	32,587	4401197858	4361889284	2302430131	4367460649
6144	42,637	96,83	14873804739	14800838124	7778378109	13967254728
8192	103,083	229,146	35039622666	35202244960	21167399112	36534415940
10240	200,211	449,326	68876206013	68743092732	68954430238	61579497457

Tempos e cache misses 8 threads

Matrix Size	Line Parallel 1	Line Parallel 2	Line Parallel 1 Cache 1	Line Parallel 1 Cache 2	Line Parallel 2 Cache 1	Line Parallel 2 Cache 2
600	0,019	0,164	3392799	7369153	8045668	32599647
1000	0,126	0,459	9378089	19634746	28711885	94520882
1400	0,265	1,213	42563883	79332691	68069943	232107402
1800	0,594	2,472	94944875	191907374	132892149	388734096
2200	1,086	5,184	259221759	347260414	230030618	594477931
2600	1,88	8,398	549656422	569019896	362186158	885345210
3000	2,957	12,64	844874483	870637139	539311646	1249914172
4096	7,674	31,712	2196317808	2212509814	1221966209	2495971898
6144	25,798	97,379	7436163519	7464170865	4045812793	7705462837
8192	65,889	228,611	17492249746	17742074684	9267401770	16092036115
10240	131,051	457,526	34356649046	34669098560	18446060765	30592088392

A2.2 Mflops

Mflops 16 threads

Simple Multiplication	Line Multiplication	Block 128	Block 256	Block 512	Line Parallel 1	Line Parallel 2	Python Simple	Python Line
1125	2138,614	Not Measured	Not Measured	Not Measured	10285,714	1009,346	14,83	9,781
834,028	2145,923	Not Measured	Not Measured	Not Measured	12987,013	1519,757	12,376	9,558
840,686	1638,209	Not Measured	Not Measured	Not Measured	13788,945	1691,739	11,407	9,412
325,919	1649,321	Not Measured	Not Measured	Not Measured	13017,857	1409,377	10,616	9,524
275,641	1649,83	Not Measured	Not Measured	Not Measured	10140,952	1434,46	10,407	9,767
245,15	1670,881	Not Measured	Not Measured	Not Measured	10600,724	1579,439	10,616	9,429
230,836	1673,9	Block 128	Block 256	Block 512	12368,301	1581,722	12,142	8,883
Not Measured	1638,831	2064,02	2440,062	2064,206	10890,567	1709,227	Not Measured	Not Measured
Not Measured	1670,543	2194,358	2482,746	2557,458	10153,142	1966,127	Not Measured	Not Measured

Not Measured	1658,539	2093,12	1952,639	1604,475	8215,243	2043,785	Not Measured	Not Measured
Not Measured	1650,806	2158,514	2479,619	2501,827	7762,12	2087,258	Not Measured	Not Measured

Mflops 8 threads

Matriz Size	Line1 Parallel1	Line1 Parallel2
600	11368,421	1317,073
1000	7936,508	2178,649
1400	10354,717	2262,16
1800	9818,182	2359,223
2200	9804,788	2054,012
2600	9348,936	2092,879
3000	9130,876	2136,076
4096	8954,845	2166,987
6144	8990,163	2381,707
8192	8343,666	2404,765
10240	8193,313	2346,843

Mflops 4 threads

Matriz Size	Line1 Parallel 1	Line1 Parallel 2
600	7714,286	1862,069
1000	6097,561	2801,12
1400	7072,165	3111,111
1800	6380,744	2712,558
2200	6130,109	1950,54
2600	6002,732	2331,344
3000	5825,243	1798,082
4096	5652,667	2108,8
6144	5439,6	2395,211
8192	5333,138	2399,151
10240	5363,051	2389,672

A2.3 Speedups

Speedup 16 threads

Matrix Size	Line Parallel 1	Line Parallel 2
600	4,81	0,472
1000	6,052	0,708
1400	8,417	1,033
1800	7,893	0,855
2200	6,147	0,869
2600	6,344	0,945
3000	7,389	0,945
4096	6,645	1,043
6144	6,078	1,177
8192	4,953	1,232
10240	4,702	1,264

Speedup 8 threads

Matrix Size	Line Parallel 1	Line Parallel 2
600	5,316	0,616
1000	3,698	1,015
1400	6,321	1,381
1800	5,953	1,43
2200	5,943	1,245
2600	5,595	1,253
3000	5,455	1,276
4096	5,464	1,322
6144	5,382	1,426
8192	5,031	1,45
10240	4,963	1,422

Speedup 4 threads

Matrix Size	Line Parallel 1	Line Parallel 2
600	3,607	0,871
1000	2,841	1,305
1400	4,317	1,899
1800	3,869	1,645

2200	3,716	1,182
2600	3,593	1,395
3000	3,48	1,074
4096	3,449	1,287
6144	3,256	1,434
8192	3,216	1,447
10240	3,249	1,448

A2.4 Eficiência

Eficiência 16 threads

Matrix Size	Line Parallel 1	Line Parallel 2
600	30,10%	2,90%
1000	37,80%	4,40%
1400	52,60%	6,50%
1800	49,30%	5,30%
2200	38,40%	5,40%
2600	39,70%	5,90%
3000	46,20%	5,90%
4096	41,50%	6,50%
6144	38,00%	7,40%
8192	31,00%	7,70%
10240	29,40%	7,90%

Eficiência 8 threads

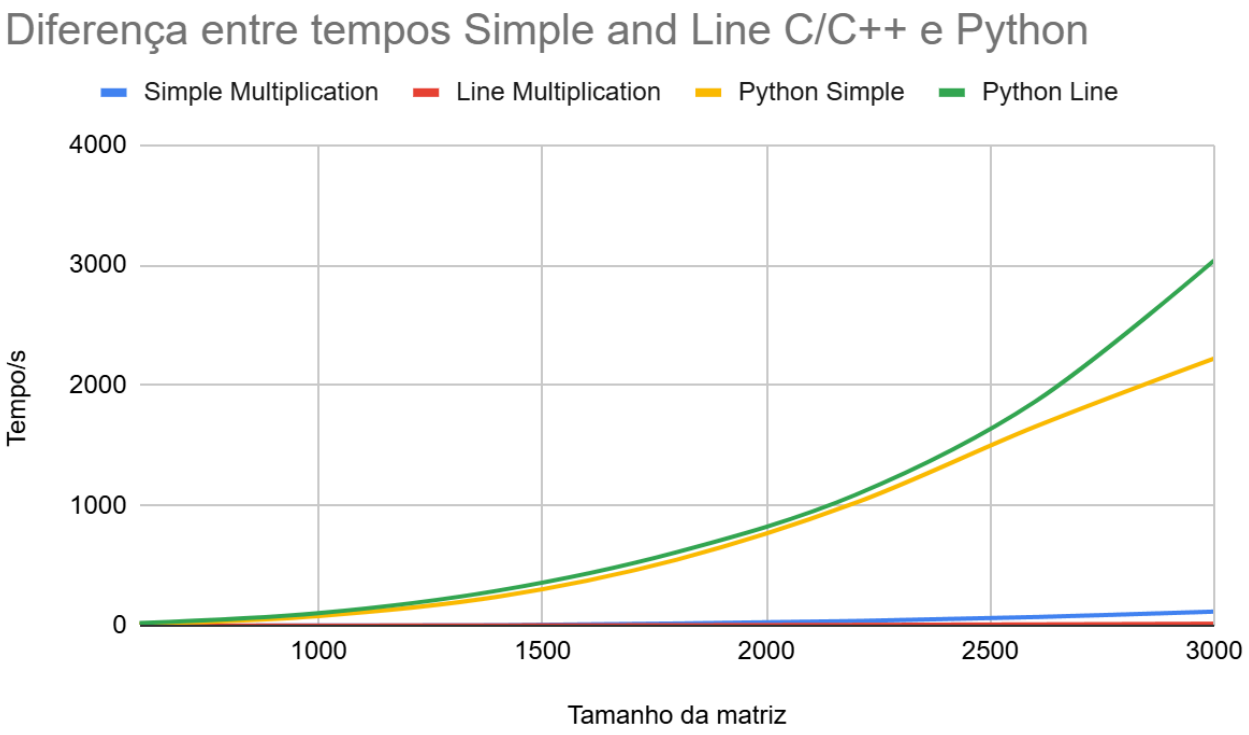
Matrix Size	Line Parallel 1	Line Parallel 2
600	66,40%	7,70%
1000	46,20%	12,70%
1400	79,00%	17,30%
1800	74,40%	17,90%

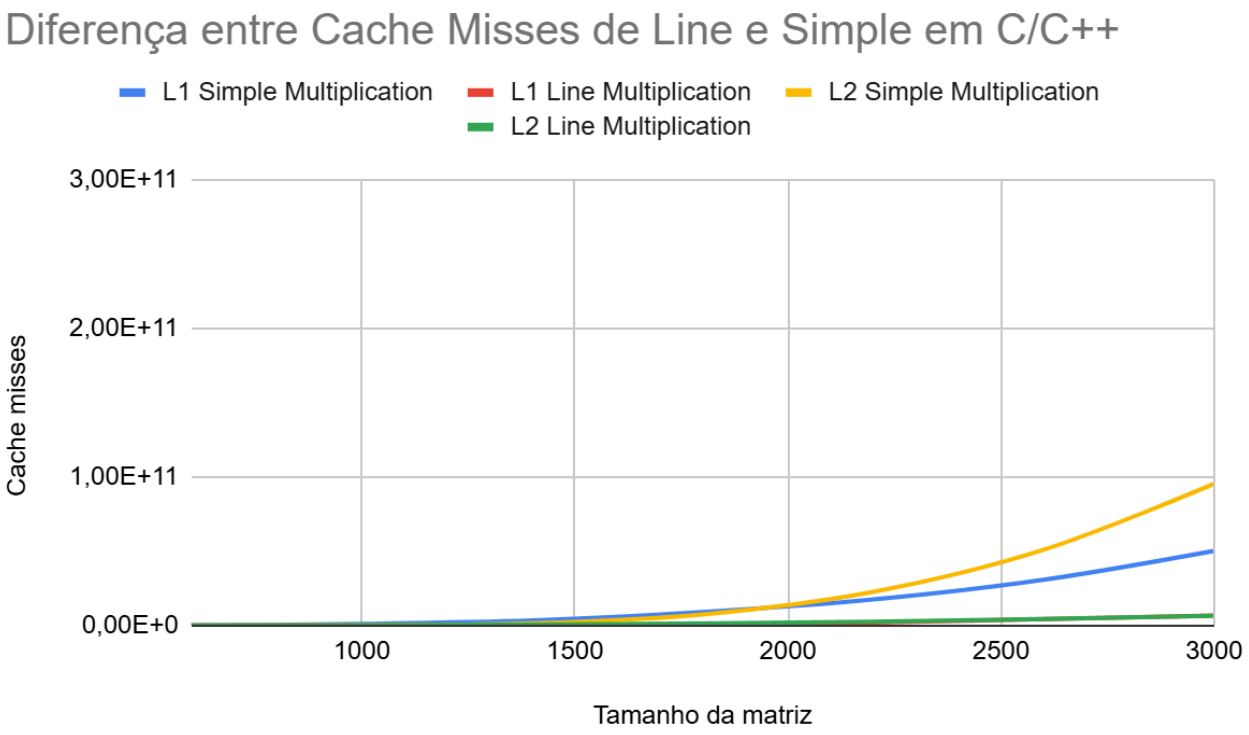
2200	74,30%	15,60%
2600	69,90%	15,70%
3000	68,20%	16,00%
4096	68,30%	16,50%
6144	67,30%	17,80%
8192	62,90%	18,10%
10240	62,00%	17,80%

Eficiência 4 threads

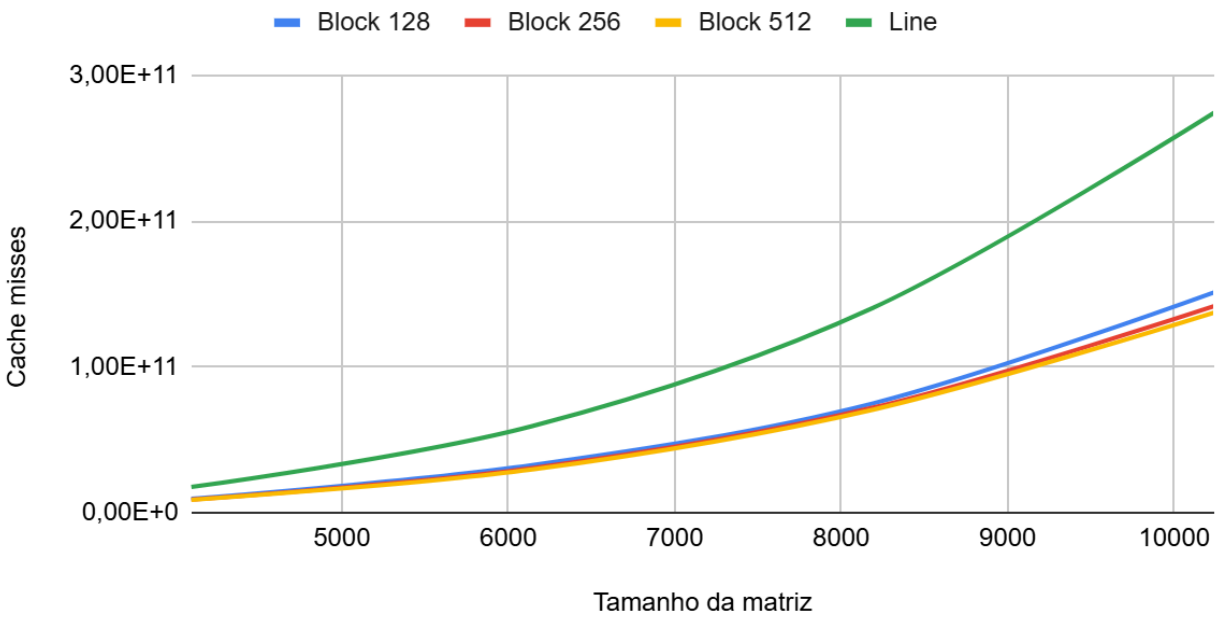
Matrix Size	Line Parallel 1	Line Parallel 2
600	90,20%	21,80%
1000	71,00%	32,60%
1400	107,90%	47,50%
1800	96,70%	41,10%
2200	92,90%	29,60%
2600	89,80%	34,90%
3000	87,00%	26,90%
4096	86,20%	32,20%
6144	81,40%	35,80%
8192	80,40%	36,20%
10240	81,20%	36,20%

A2.5 Gráficos

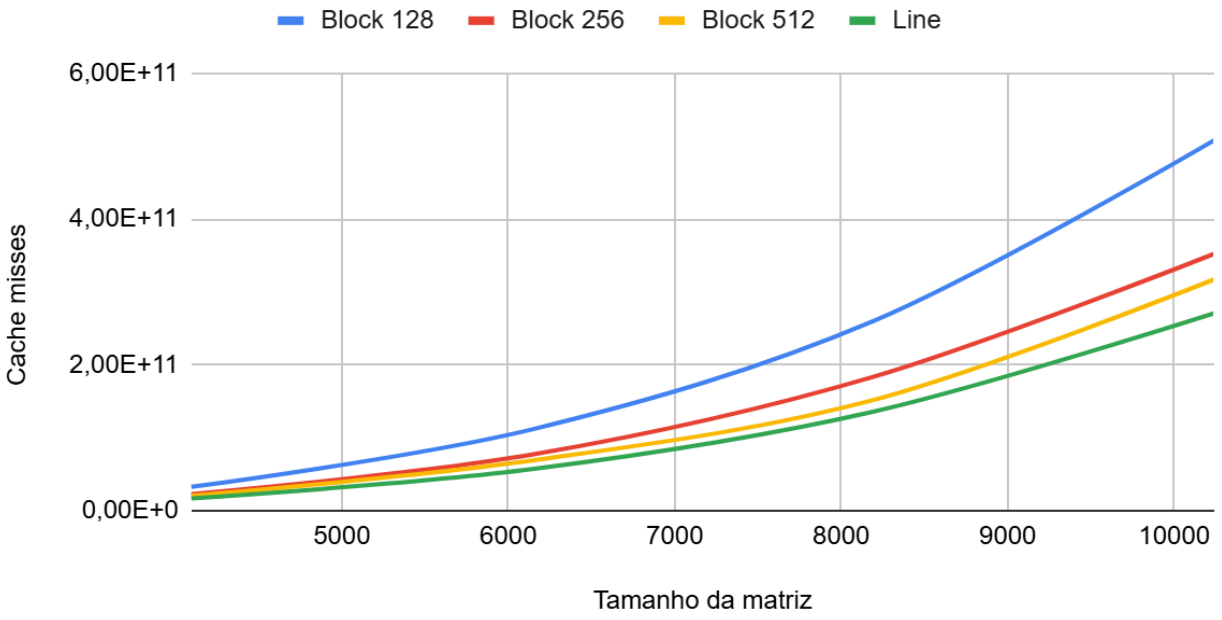

Diferença entre tempos Simple and Line C/C++ e Python
Simple Multiplication
Line Multiplication
Python Simple
Python Line
4000
3000
2000
1000
0
Tempo/s
1000
1500
2000
2500
3000
Tamanho da matriz


Diferença entre Cache Misses de Line e Simple em C/C++
L1 Simple Multiplication
L1 Line Multiplication
L2 Simple Multiplication
L2 Line Multiplication
3,00E+11
2,00E+11
1,00E+11
0,00E+0
Cache misses
1000
1500
2000
2500
3000
Tamanho da matriz

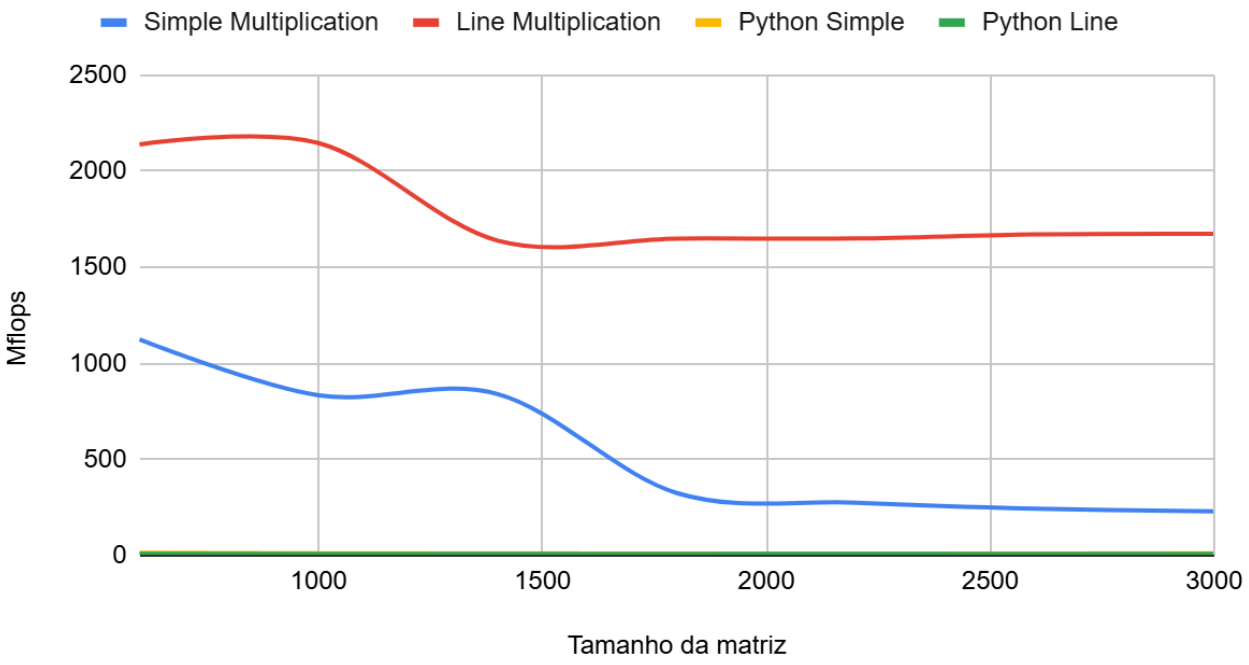
L1 cache misses com Tamanhos de Bloco Diferentes e Linha Mult



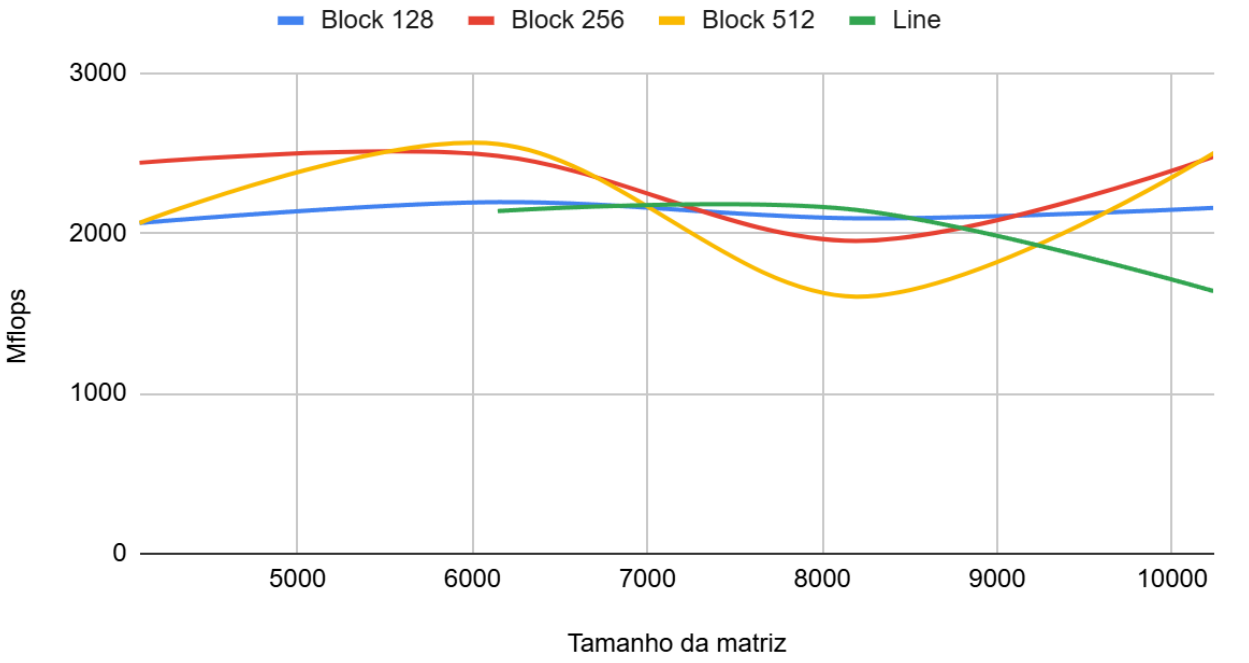
L2 cache misses com Tamanhos de Bloco Diferentes e Linha Mult

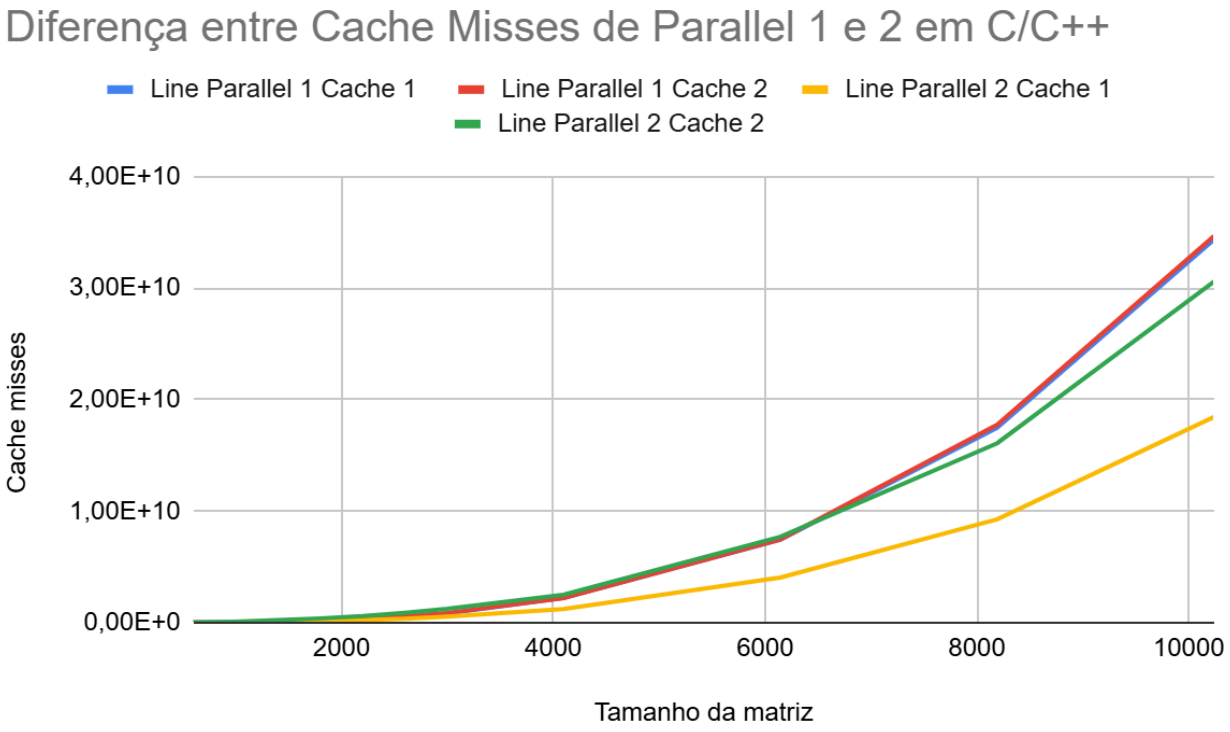


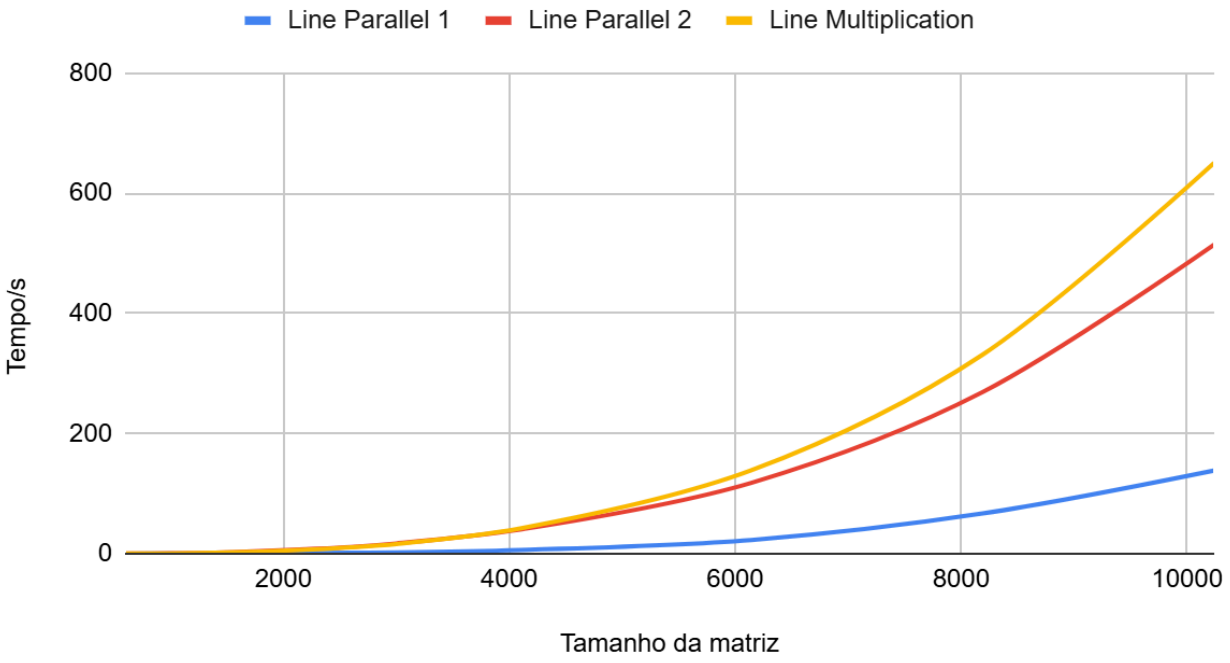
MFLOPS Simples e Linha

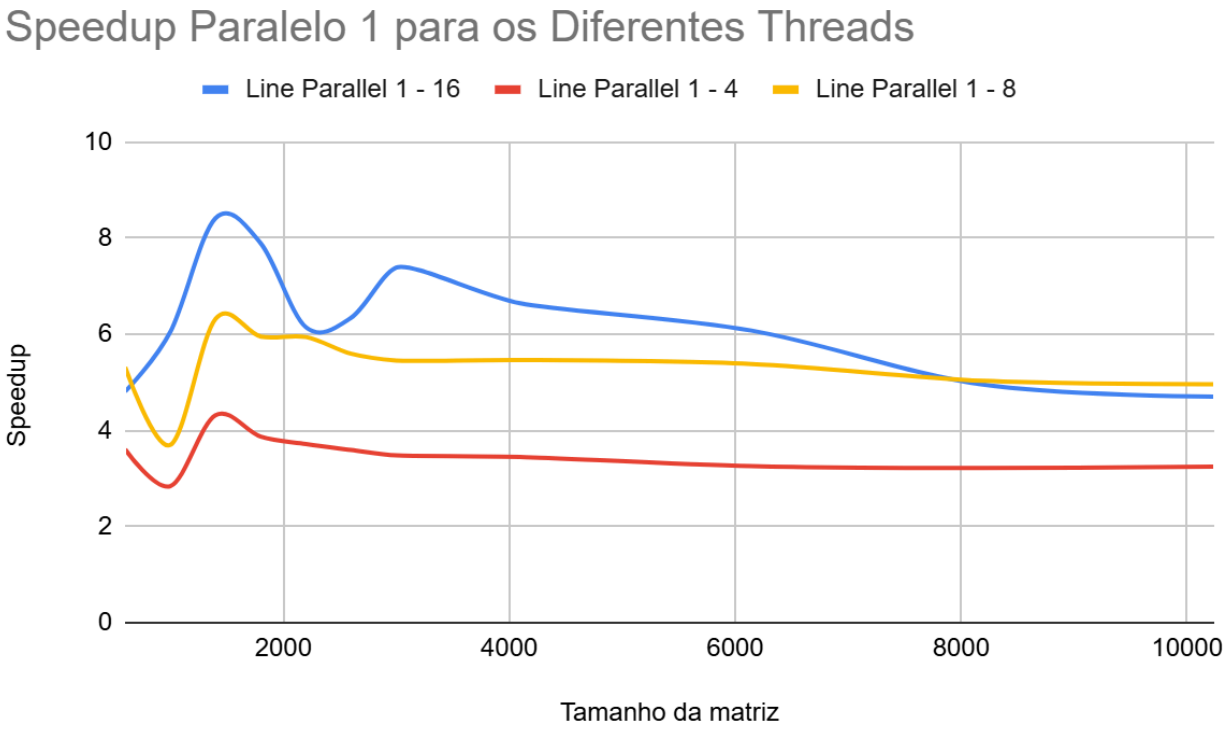


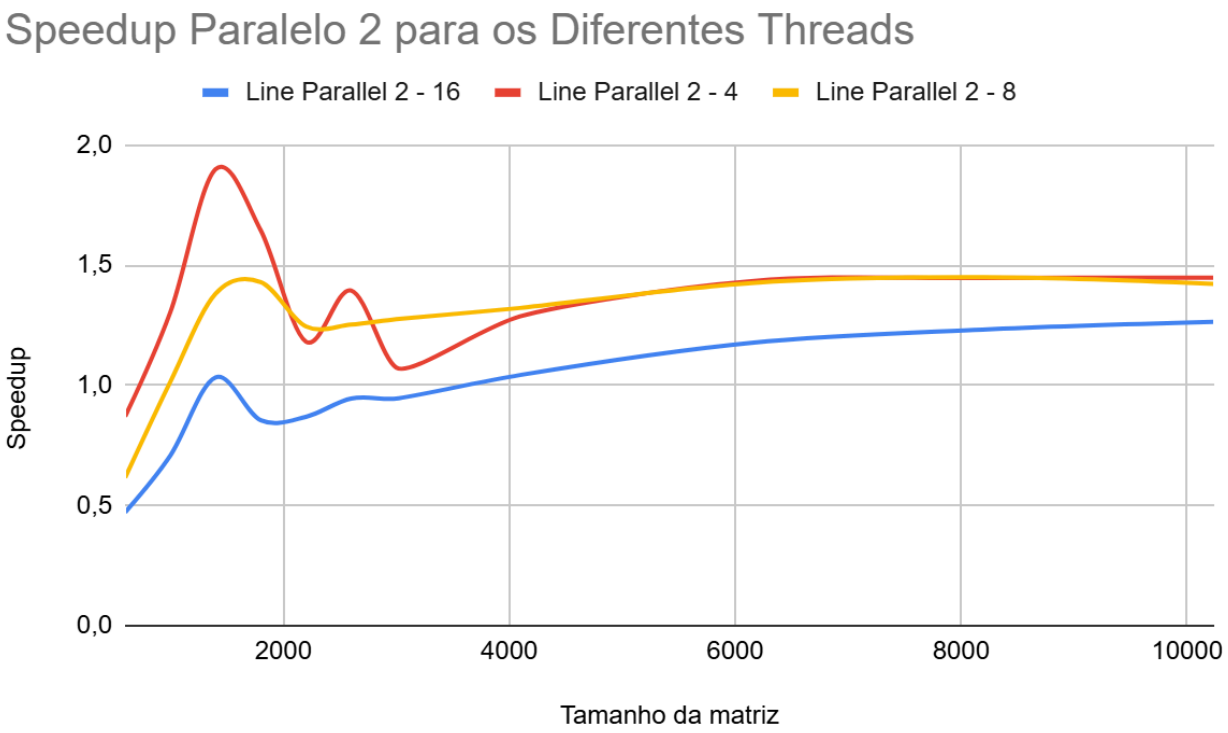
MFLOPS Matriz por Bloco

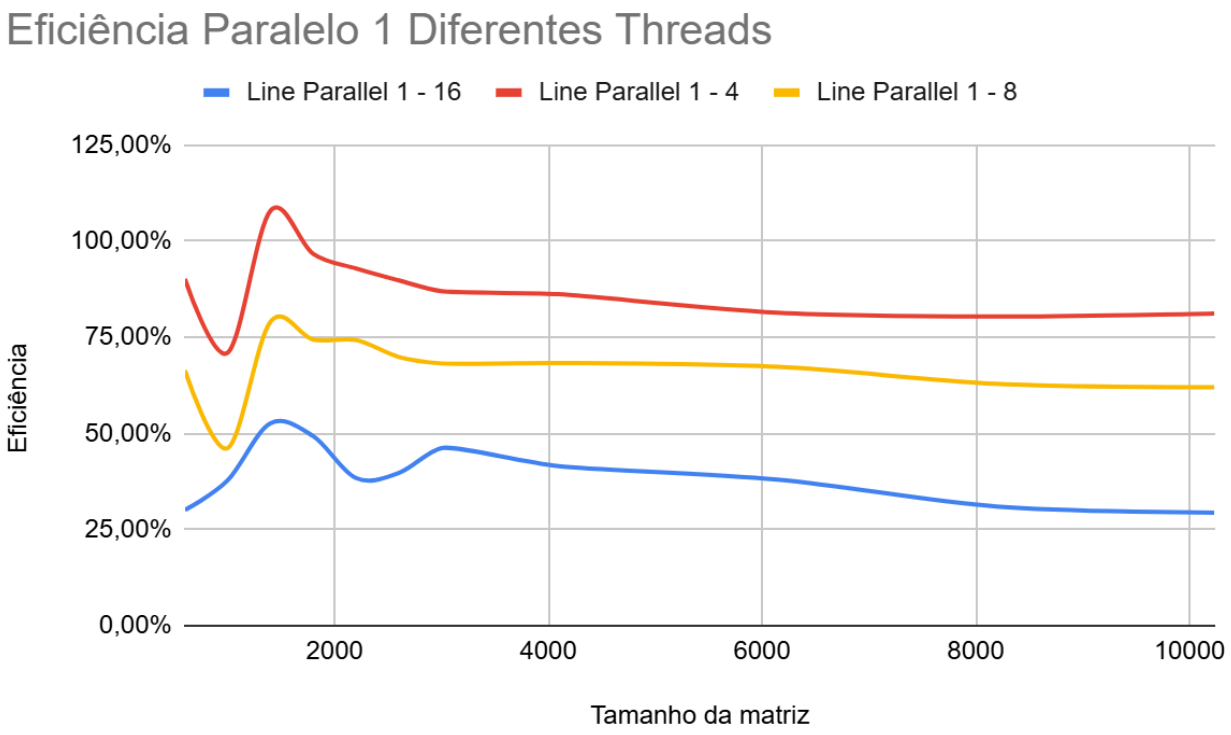


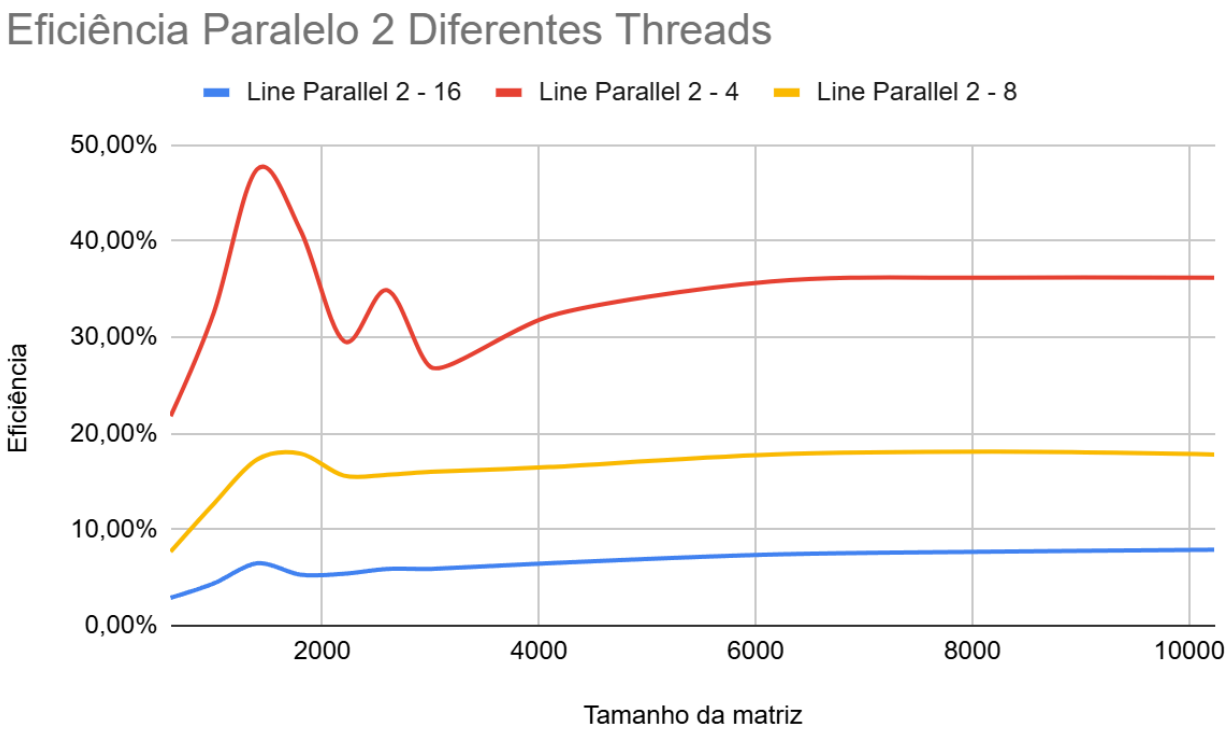

Diferença entre Cache Misses de Parallel 1 e 2 em C/C++
Line Parallel 1 Cache 1
Line Parallel 1 Cache 2
Line Parallel 2 Cache 1
Line Parallel 2 Cache 2
4,00E+10
3,00E+10
2,00E+10
1,00E+10
0,00E+0
Cache misses
2000
4000
6000
8000
10000
Tamanho da matriz

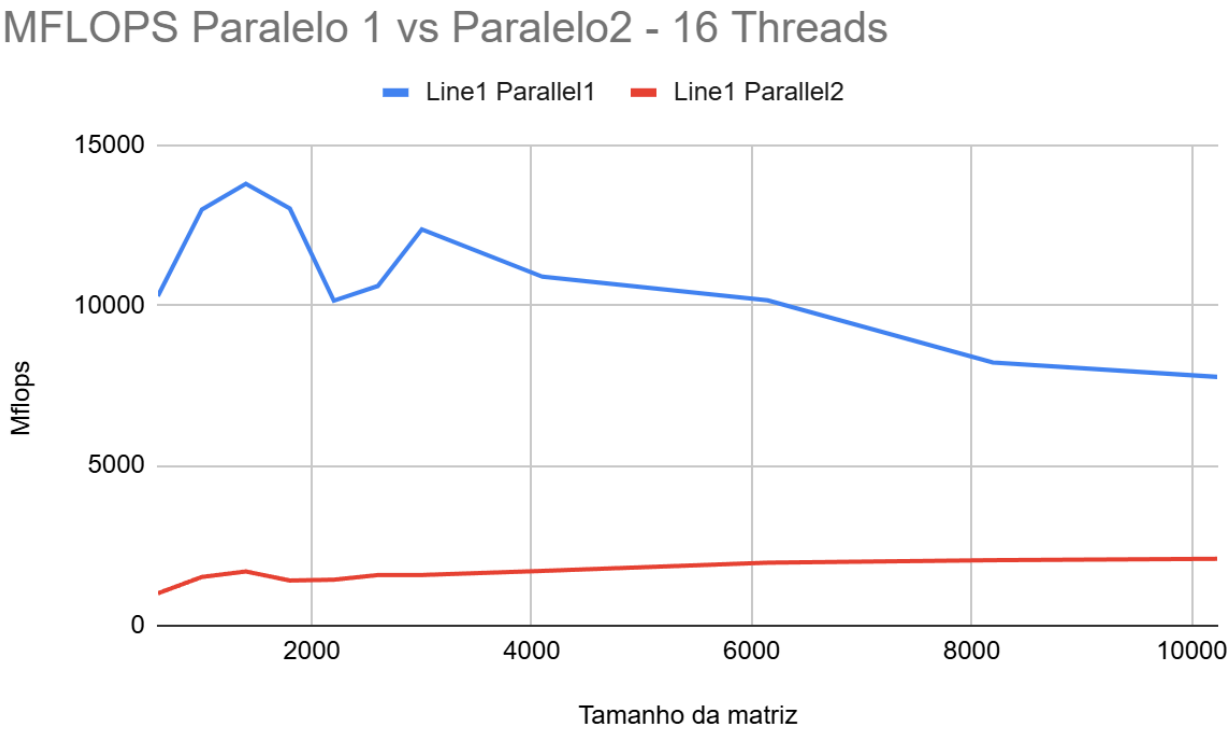

Diferentes Tempos Linha / Paralelos
Line Parallel 1
Line Parallel 2
Line Multiplication
800
600
400
200
0
Tempo/s
2000
4000
6000
8000
10000
Tamanho da matriz

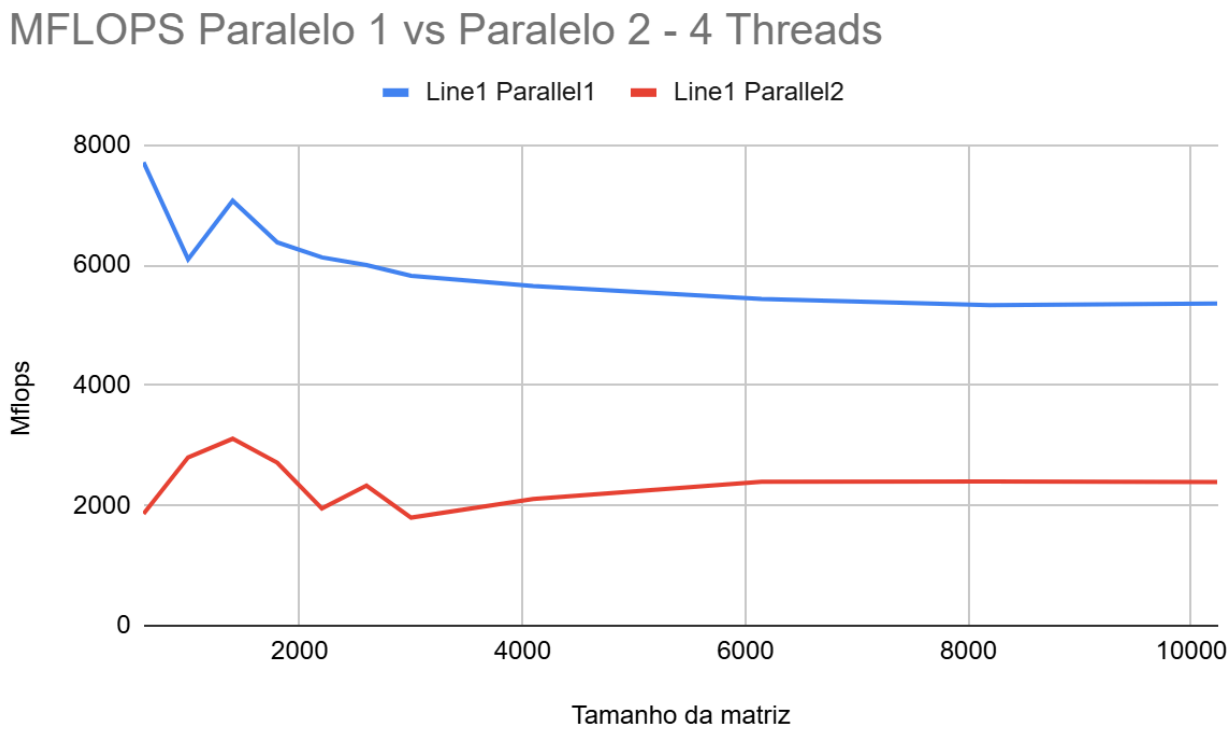

Speedup Paralelo 1 para os Diferentes Threads
Line Parallel 1 - 16
Line Parallel 1 - 4
Line Parallel 1 - 8
10
8
6
4
2
0
Speedup
2000
4000
6000
8000
10000
Tamanho da matriz

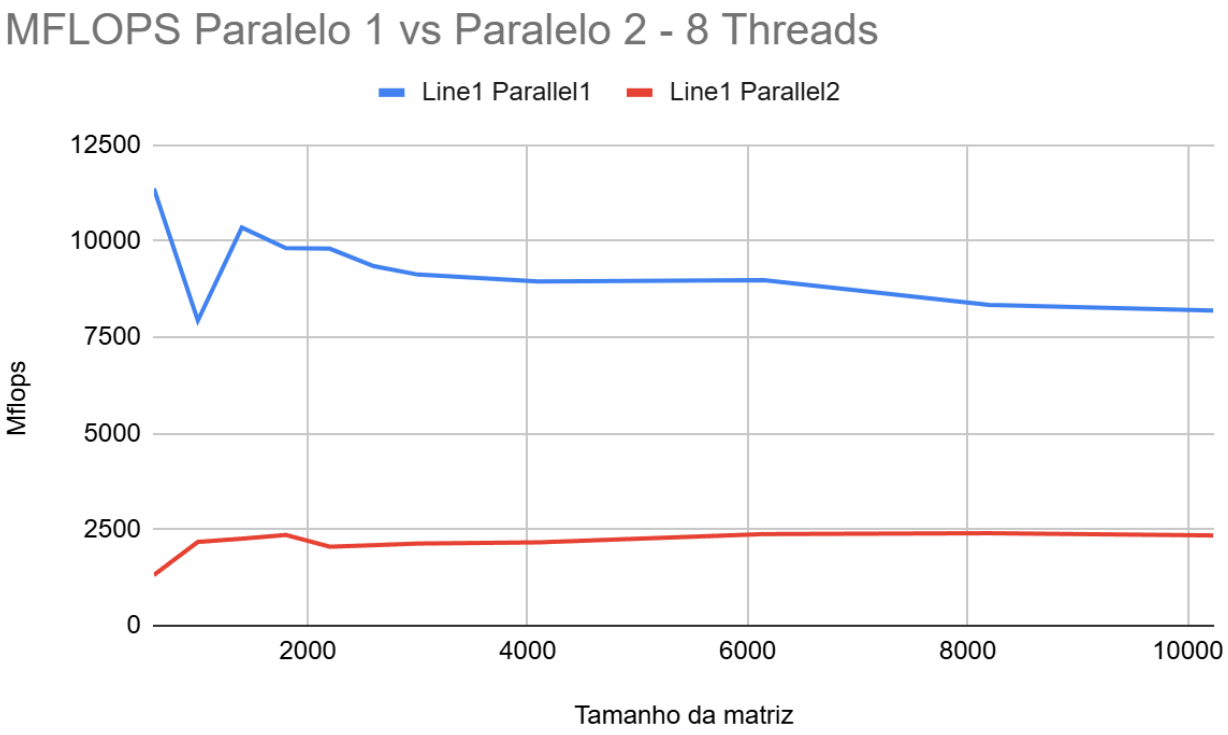

Speedup Paralelo 2 para os Diferentes Threads
Line Parallel 2 - 16
Line Parallel 2 - 4
Line Parallel 2 - 8
2,0
1,5
1,0
0,5
0,0
Speedup
2000
4000
6000
8000
10000
Tamanho da matriz


Eficiência Paralelo 1 Diferentes Threads
Line Parallel 1 - 16
Line Parallel 1 - 4
Line Parallel 1 - 8
125,00%
100,00%
75,00%
50,00%
25,00%
0,00%
Eficiência
2000
4000
6000
8000
10000
Tamanho da matriz


Eficiência Paralelo 2 Diferentes Threads
Line Parallel 2 - 16
Line Parallel 2 - 4
Line Parallel 2 - 8
50,00%
40,00%
30,00%
20,00%
10,00%
0,00%
Eficiência
2000
4000
6000
8000
10000
Tamanho da matriz


MFLOPS Paralelo 1 vs Paralelo2 - 16 Threads
Line1 Parallel1
Line1 Parallel2
15000
10000
5000
0
Mflops
2000
4000
6000
8000
10000
Tamanho da matriz


MFLOPS Paralelo 1 vs Paralelo 2 - 4 Threads
Line1 Parallel1
Line1 Parallel2
8000
6000
4000
2000
0
Mflops
2000
4000
6000
8000
10000
Tamanho da matriz


MFLOPS Paralelo 1 vs Paralelo 2 - 8 Threads
Line1 Parallel1
Line1 Parallel2
12500
10000
7500
5000
2500
0
Mflops
2000
4000
6000
8000
10000
Tamanho da matriz