MAC0219 - Relatório Mini-EP 6

Nathália Yukimi Uchiyama Tsuno (14600541)

October 2024

1 Um Bloco de Cada Vez

A fim de se buscar a blocagem com um desempenho satisfatório, produzimos alguns experimentos para tamanhos de blocos diferentes. Abaixo, pode-se encontrar os resultados, em conjunto com suas estatísticas e intervalos de confiança, além de um gráfico ilustrativo.

Para esses experimentos, consideramos que a matriz é quadrada e de dimensões múltiplas a 2. Em especial, para uma matriz de tamanho 2048.

N	N = 4	N = 8	N = 16	N = 32	N = 64	N = 128
1	4.203487	3.846327	3.694923	3.790498	3.689688	3.665526
2	4.357173	3.841002	3.684741	3.717751	3.583194	3.785476
3	4.568433	4.015357	3.680335	3.684005	3.73003	3.573829
4	4.40901	3.986778	3.674698	4.089555	3.664492	3.848167
5	4.454396	3.992641	3.686134	3.700896	3.678558	3.718892
6	4.430011	4.076105	3.67702	3.736725	3.804004	3.658894
7	4.310167	3.989631	3.671149	3.826714	3.642858	3.674463
8	4.632094	4.073597	3.808802	3.604762	3.65911	3.640172
9	4.374932	3.967247	3.809811	3.678975	3.631322	3.674235
10	4.286625	4.289054	3.826857	3.701674	3.707832	3.55768

Tabela 1: Tempo de execução, em segundos, dado o tamanho do bloco (N)

Com mais detalhes, tenha a distribuição estatística:

Iteração	Mediana	Média	Desvio Padrão	Mínimo	Máximo	Q0	Q1	Q3	Q4
4	4.391971	4.402633	0.128285	4.203487	4.632094	4.203487	4.321918	4.448300	4.632094
8	3.991136	4.007774	0.126797	3.841002	4.289054	3.841002	3.972130	4.059037	4.289054
16	3.685437	3.721447	0.065173	3.671149	3.826857	3.671149	3.677849	3.780332	3.826857
32	3.709713	3.753155	0.132991	3.604762	4.089555	3.604762	3.688228	3.777055	4.089555
64	3.671525	3.679109	0.060129	3.583194	3.804004	3.583194	3.646921	3.703296	3.804004
128	3.669880	3.679733	0.087849	3.557680	3.848167	3.557680	3.644853	3.707785	3.848167

Tabela 2: Distribuição dos dados obtidos, em segundos

E, enfim, os intervalos de confiança:

Tamanho do Bloco	Intervalo de Confiança
4	$[4.32312262s, 4.48214298s]$
8	$[3.92918578s, 4.08636202s]$
16	$[3.68105318s, 3.76184082s]$
32	$[3.67072821s, 3.83558279s]$
64	$[3.64184086s, 3.71637674s]$
128	$[3.62528512s, 3.73418168s]$

Tabela 3: Intervalos de Confiança

Para uma interpretação gráfica, segue um gráfico das médias:

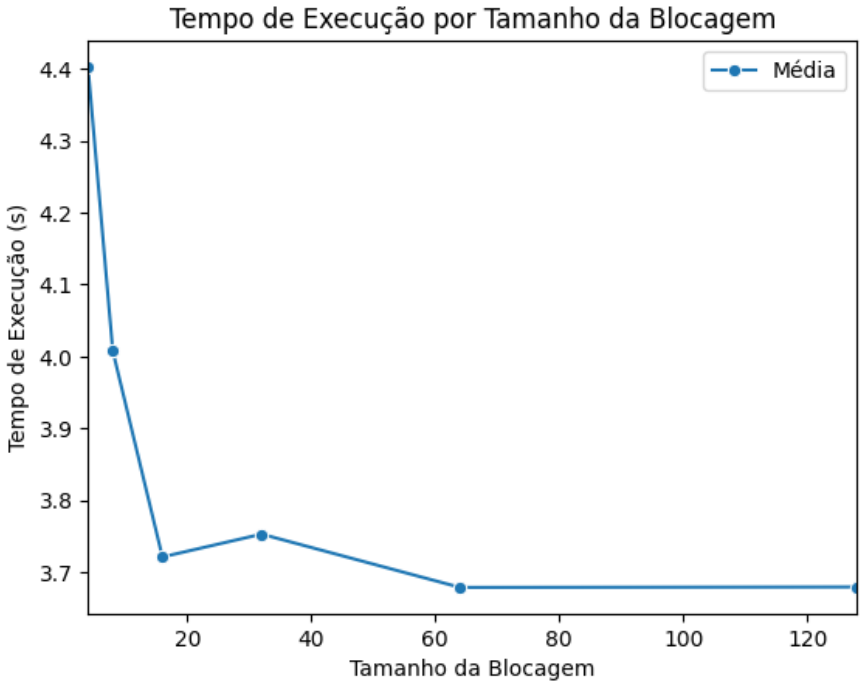

Figura 1: Distribuição da Média dos Tempos de Execução, em segundos

Assim, para a melhor execução da blocagem, é interessante o uso para N = 64.

2 A Ordem dos Fatores Altera O Produto

Para responder a primeira pergunta: "Mostre, com embasamento estatístico, a variação do tempo entre as funções `matrix_dgemm_0`, `matrix_dgemm_1` e `matrix_dgemm_2` . Explique o porquê.", vamos executar alguns experimentos.

Executando o binário `main.c`, iteramos os programas `.\main -matrix-size i -algorithm j` 10 vezes, para $i \in [1, 4096], j \in \{0, 1, 2\}$. Os resultados puderam gerar esse gráfico em escala real:

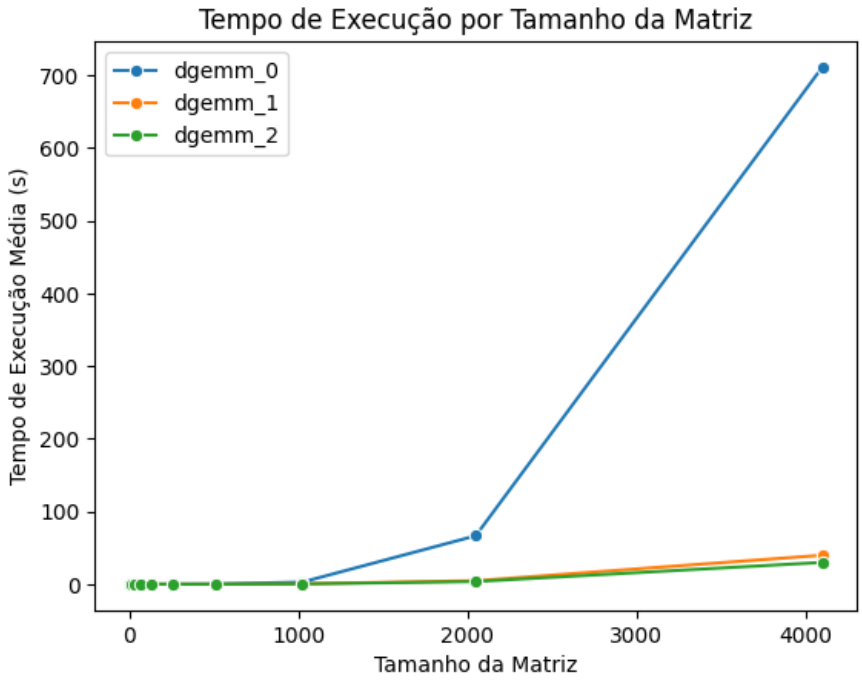

Figura 2: Relação entre os tamanhos da matriz e os tempos médios de execução, em segundos

Vale relembrar que estamos supondo tamanhos de matrizes múltiplas à potências de dois.

Ademais, tem-se as distribuições estatísticas para esses dados:

Tamanho da Matriz	Mediana	Média	Desvio Padrão	Mínimo	Máximo	Q0	Q1	Q3	Q4
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
64	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
128	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
256	0.01	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.01
512	0.25	0.25	0.01	0.24	0.27	0.24	0.25	0.25	0.27
1024	2.73	2.87	0.54	2.46	4.32	2.46	2.60	2.84	4.32
2048	65.37	66.52	4.51	62.64	78.06	62.64	64.06	66.67	78.06
4096	703.08	710.83	17.84	693.95	744.07	693.95	699.23	720.75	744.07

Tabela 4: Distribuição de tempo de execução, em segundos, para `dgemm_0`

Tamanho da Matriz	Mediana	Média	Desvio Padrão	Mínimo	Máximo	Q0	Q1	Q3	Q4
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
64	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
128	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
256	0.01	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.01
512	0.05	0.05	0.00	0.05	0.06	0.05	0.05	0.05	0.06
1024	0.51	0.53	0.04	0.50	0.60	0.50	0.50	0.55	0.60
2048	4.69	4.72	0.22	4.49	5.11	4.49	4.56	4.79	5.11
4096	39.49	39.80	1.18	38.69	42.66	38.69	39.20	39.94	42.66

Tabela 5: Distribuição de tempo de execução, em segundos, para `dgemm_1`

Tamanho da Matriz	Mediana	Média	Desvio Padrão	Mínimo	Máximo	Q0	Q1	Q3	Q4
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
32	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
64	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
128	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
256	0.01	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.01
512	0.06	0.06	0.00	0.06	0.06	0.06	0.06	0.06	0.06
1024	0.44	0.45	0.02	0.44	0.49	0.44	0.44	0.46	0.49
2048	3.60	3.64	0.13	3.54	3.97	3.54	3.57	3.65	3.97
4096	29.78	29.93	0.57	29.31	30.94	29.31	29.56	30.14	30.94

Tabela 6: Distribuição de tempo de execução, em segundos, para `dgemm_2`

O programa original, com os laços encaixados na ordem `i, j, k`, demonstram um crescimento mais acentuado, que se torna mais evidente, graficamente, a $N = 2048$, mas, que se inciam a $N = 512$, pela tabela. Isso se justifica pelo argumento da Álgebra Linear. Essencialmente, cada posição de $C_{i,j}$ é oriunda do produto escalar entre os elementos da i-ésima linha de A e a j-ésima de B. Portanto, nada mais justo do que iterar cada elemento da i-ésima linha de A e de cada elemento da j-ésima coluna de B, para se calcular $C_{i,j}$. A termos de corretude e sob uma visão puramente teórica, `.\main -matrix-size i -algorithm 0` é um algoritmo ideal. Contudo, o problema emerge quando se traz essa problemática para o mundo real e as limitações físicas do Hardware.

Para cada elemento requisitado na matriz X, para $X \in \{A, B, C\}$, a CPU carrega os dados na L1-Cache este e mais cerca dos próximos 64 bytes, caso haja um cache-miss. Logo, A aproveita bem o ideal de se haver uma memória cache para minimizar o tempo de carregamento de dados da RAM para os registradores. Por outro lado, a depender do tamanho da matriz, B recebe um cache miss a cada iteração, pois, calcula todas as linhas da j-ésima coluna. Assim, a distância do elemento atual para o próximo pode ser mais de 64 bytes, desfavorecendo a existência da memória cache e exigindo um carregamento constante na RAM.

Portanto, no gráfico, notamos que para valores de matriz cada vez maiores, o impacto do cache miss é mais evidente, atrapalhando o tempo de alocação de dados e processamento, gerando esse crescimento expressivo. O programa `dgemm_0` contém as maiores médias, mas, em conjunto, aos maiores desvios padrões.

O programa com a modificação apenas sob a ordem dos laços para `i, k, j` também sofre um crescimento cúbico em razão da natureza do algoritmo que é da ordem $\theta(N^3)$, pois, note que para $N = 4096$, de $N = 2048$, o crescimento da entrada dobrou, mas, o tempo $log_2(39.8/4.72) = 3.076$, triplicou. Contudo, em relação ao programa original, este possuí um desempenho consideravelmente melhor. Para $N = 4096$, a melhora foi de $\frac{710.83}{39.80} = 17.8601$.

Apesar de matemáticamente igualmente corretos, este algoritmo se diferencia pelo aproveitamento em se bem utilizar o potencial do cache. A leitura de A permanece a mesma, linear sobre a i-ésima linha. A diferença é que, para o k-ésimo elemento da i-ésima linha de A, o algoritmo faz a leitura dos elementos na k-ésima linha de B e, depois vai somando com os próximos elementos das próximas linhas. Note que, algebricamente, o cálculo da i-ésima linha de C depende do produto da i-ésima linha de A com toda a matriz B. Assim, utilizando bem a k-ésima linha de B e acrescendo posteriormente, conseguimos um melhor aproveitamento do cache e garantimos maior velocidade e reduzindo os acessos à RAM.

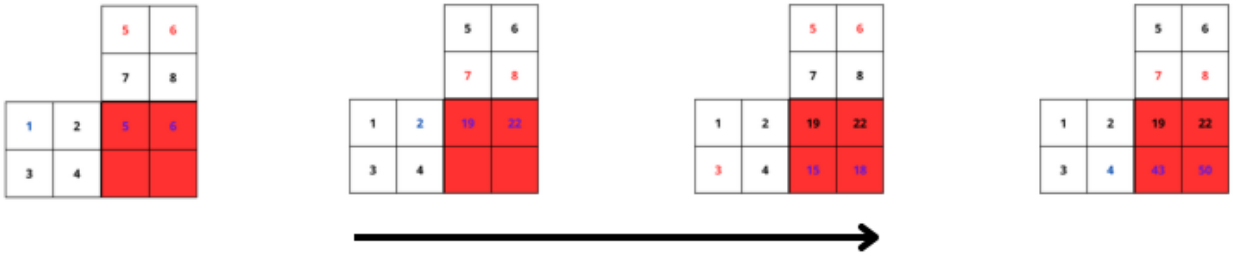

Figura 3: Processo para uma matriz 2x2

Finalmente, o programa `dgemm_2`, o mais veloz de todos para valores expressivamente maiores. Essa diferenciação começa para $N = 2048$, embora para $N = 1024$, a blocagem seja mais ineficiente do que para sem ela. Nesta situação temos, para $N = 4096$, um programa cerca de 10s mais rápido do que sem a blocagem. Vale contestar também que o desvio padrão é menor para este caso do que para os outros.

Neste caso, queremos maximizar a utilização do cache para o cálculo sobre as colunas de B. Para matrizes expressivamentes grandes, essa operação pode resultar numa frequência considerável de cache misses (embora em menor escala do que a versão original), a depender do tamanho da linha operada. Assim, respondemos à pergunta "Como você usou a blocagem para melhorar a velocidade da multiplicação de matrizes?". Logo, podemos fazer uma blocagem sobre as partições da matrizes multiplicadoras.

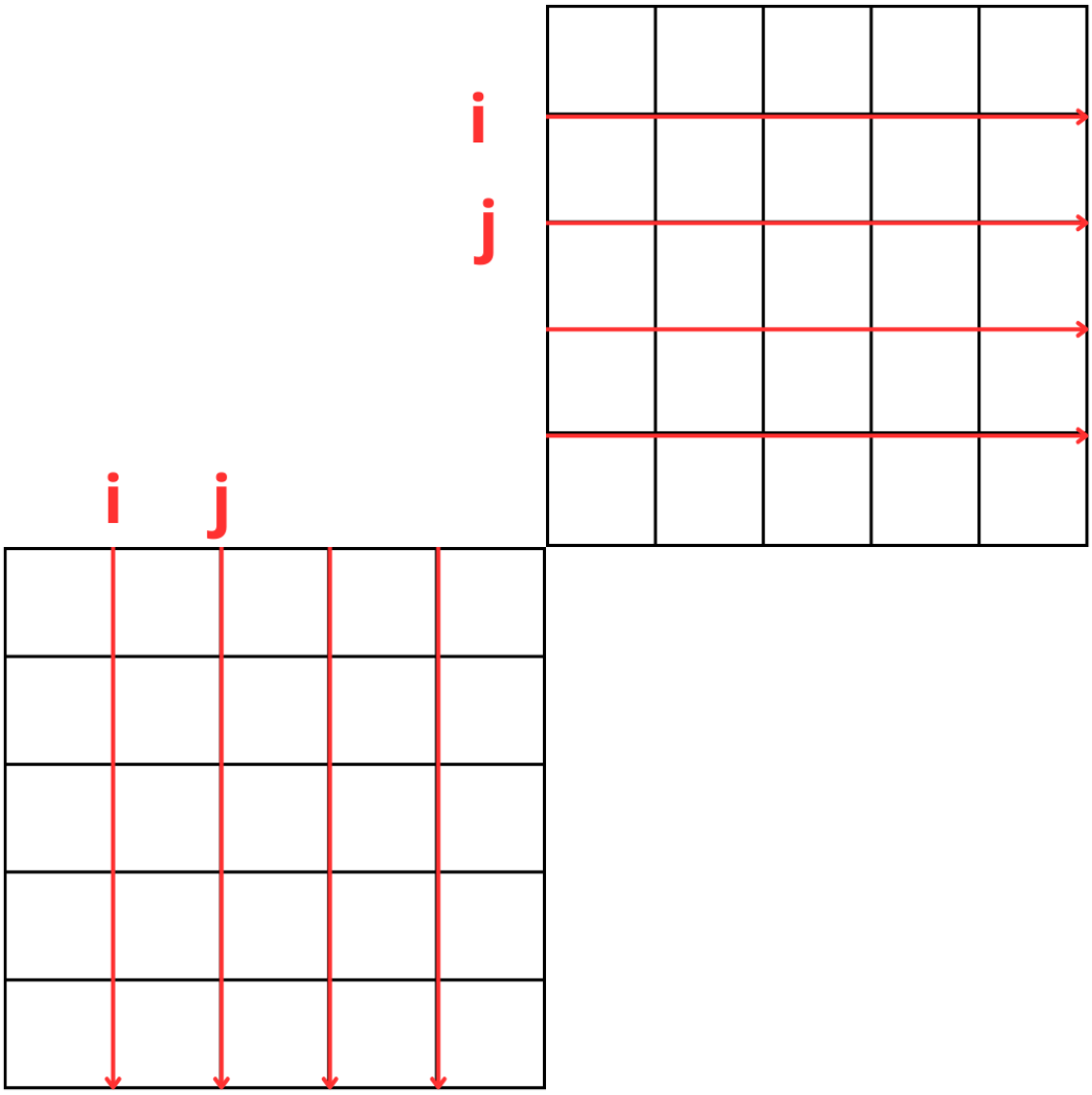

Figura 4: Blocagem

No método da blocagem, a multiplicação se restringe no intervalos de blocos pré-definidos de A e de B. Ao fim, o bloco se desloca para a esquerda e o processo multiplicativo recomeça.

Apesar de reduzirmos a capacidade do uso de cache do bloco A, o método da blocagem beneficia a iteração pelo bloco B, pois reduz quantidade de dados a serem inseridos no cache por vez. Lembre-se que o método aplicado pela inversão dos laços define a multiplicação $A_{i,*}xB$, isto é, o produto matricial da i-ésima linha de A, com B. Assim, a depender do tamanho B, haveria muitos cache misses, levando a uma maior busca à RAM.

Portanto, o método da blocagem reduz a quantidade de elementos a serem operados por vez, sobretudo aos elementos de B.