Universidade Estadual de Santa Cruz
Davi Roriz, Estêvão Sousa.

Relatório I:Análise de multiplicação de Matrizes bidimensionais utilizando OpenMP

Ilhéus - Bahia
2025

Davi Roriz, Estêvão Sousa

Relatório I:Análise de multiplicação de Matrizes bidimensionais utilizando OpenMP

Relatório apresentado à matéria Processamento Paralelo do curso de Ciências da Computação da Universidade Estadual de Santa Cruz, como avaliação do primeiro crédito da matéria.

Prof. Orientador: Prof. Esbel Tomás Valero.

Ilhéus - Bahia
2025

1. INTRODUÇÃO

1.1 Objetivo do projeto

Este projeto tem como objetivo principal a implementação e análise de desempenho de uma versão customizada do algoritmo DGEMM (Double Precision General Matrix Multiply), abordando tanto o desenvolvimento computacional quanto a análise de desempenho.

Para tal, foi necessário uso da linguagem C++ para o desenvolvimento do algoritmo customizado dgemm. Além disso, utilizamos python para gerar tabelas e gráficos para fazermos nossa análise comparativa de desempenho entre o modelo sequencial e paralelizado com diferentes threads

1.2 Conceitos teóricos relevantes

Para a compreensão deste projeto, são fundamentais alguns conceitos teóricos relacionados ao processamento paralelo e à otimização de operações matriciais. O próprio conceito do DGEMM (*Double Precision General Matrix Multiply*) refere-se à multiplicação de matrizes de dupla precisão, operação amplamente utilizada em aplicações científicas e de engenharia, sendo necessário ser revisado.

A paralelização com OpenMP (*Open Multi-Processing*) permite a execução concorrente de partes do código em múltiplos núcleos de CPU, utilizando diretivas de compilação para gerenciar threads de forma simplificada.

Outro conceito crucial é o de blocagem (tiling), técnica que divide a matriz em blocos menores para melhor aproveitamento da hierarquia de cache, reduzindo o tempo de acesso à memória.

A ordem de loops também influencia diretamente a localidade dos dados e o desempenho do algoritmo, uma vez que afeta o padrão de acesso à memória. Por fim, as métricas de análise deste relatório: speedup (ganho de tempo) e eficiência (proporção do speedup em relação ao número de threads) são essenciais para avaliar o impacto da paralelização.

1.3 Importância da análise de desempenho

A análise de desempenho é uma etapa essencial no desenvolvimento de aplicações que demandam alto poder de processamento. Permite identificar gargalos, validar otimizações e assegurar que os recursos de hardware sejam explorados de maneira eficiente. Além disso, a análise comparativa de métricas como tempo de execução, speedup e eficiência traz uma boa visão sobre o uso eficiente do hardware junto da solução. Em ambientes científicos e industriais, a análise de desempenho é indispensável para fundamentar decisões relacionadas à implementação e otimização de sistemas críticos ou intensivos.

2. METODOLOGIA

A implementação foi realizada em linguagem C++, escolhida por conta de sua eficiência em operações de baixo nível, otimização de memória e a compatibilidade com a biblioteca OpenMP para realizar o paralelismo.

O código fonte, disponível no repositório do github, foi nomeado como “main.cpp”, que se encontra na pasta “Trabalho01”. O programa possui as funções dgemm em suas versões sequencial e paralela, uma função que gera uma matriz NxN de forma aleatória composta por 0 e 1, além de duas funções para calcular o tempo de execução de cada versão da dgemm. Os resultados são apresentados no arquivo “resultados.csv” em formato de tabela, com as informações do tamanho da matriz, tempo sequencial, tempo paralelo, speedup (temp_par / temp_seq) e eficiência (speedup / num_threads) para cada quantidade de threads utilizada. Os comandos para compilação e execução estão comentados no código fonte.

Para melhorar a otimização geral foram utilizadas algumas técnicas, como o uso do qualificador __restrict__ em ponteiros para informar ao compilador que não há aliasing (sobreposição de memória), além da inicialização das matrizes A e B com valores aleatórios usando mt19937 (Mersenne Twister) para geração pseudo-aleatória rápida e uniforme, evitando o uso de rand() que é mais lento e menos uniforme.

O código foi compilado com flags como -O3 (otimização nível 3), -march=native (otimização para a arquitetura nativa do processador), -mfma (habilita fused multiply-add para operações mais rápidas) e -fopenmp (suporte a OpenMP), garantindo que o compilador (g++ do GCC) gere código eficiente. Foi ainda testado a flag de otimização -Ofast, que supostamente seria mais rápida que a -O3, porém foi constatado um tempo de execução superior em relação a flag utilizada.

2.1 Implementação Sequencial

A versão sequencial foi implementada na função dgemm_seq, que recebe como parâmetros os ponteiros para as matrizes A, B e C, além do tamanho N da matriz quadrada.

Para lidar com matrizes grandes, foi implementada a técnica de blocagem (ou tiling) com tamanho de bloco fixo de 128, outros valores foram testados, porém o que obteve a melhor média de tempo foi o tamanho escolhido. Isso divide a multiplicação em subproblemas menores que cabem melhor nas caches do processador. Vale destacar que se o tamanho da matriz for menor ou igual a 128, a função irá executar a multiplicação normalmente, porém sem utilizar a técnica de blocagem, contudo, como o tamanho da matriz não é muito grande, a técnica daria pouco ou nenhum ganho de desempenho nesses casos.

Foi escolhida a ordem de loops ikj ao invés da ordem ijk convencional, essa decisão foi motivada pela melhoria na localidade de cache, visto que na ordem ikj o acesso à matriz B é por linhas (acesso contínuo em memória), e o acesso a C é

sequencial dentro de cada iteração interna, reduzindo misses de cache. A ordem ijk, por outro lado, acessa B por colunas, o que é menos eficiente para a aplicação.

2.2 Implementação paralela

A versão paralela foi implementada na função dgemm_par, que estende a versão sequencial adicionando paralelismo via OpenMP. Mantivemos a mesma estrutura de blocagem e a ordem ikj nos loops para preservar as otimizações de cache, mas distribuímos o trabalho entre as threads. O número de threads é passado como parâmetro e configurado com o omp_set_num_threads(num_threads) antes do bloco paralelo.

Foi utilizada a diretiva #pragma omp parallel, que indica que o bloco sintático associado deve ser executado em paralelo, criando um grupo de threads que executam o código dentro do bloco. Aplicada imediatamente após o parallel, a #pragma omp for distribui as iterações do loop entre as threads de forma estática (default).

Não utilizamos a diretiva barrier pois o fim do loop for implica em uma barreira implícita, sincronizando as threads naturalmente. Além disso, não há necessidade da diretiva ordered para criar uma ordem entre as interações. Foi testada a diretiva schedule com diferentes valores para o type, contudo, foi notado uma perda de desempenho ao usá-la, portanto foi removida da versão final do código

2.3 Hardware utilizado nos testes

Notebook 1

Componente	Especificação principal	Detalhes
Processador (CPU)	AMD Ryzen 5 5500u	6 cores, 12 threads. Frequência Base: 2.1GHZ
RAM	12Gb 3200MHZ	2Gb compartilhados para placa de vídeo integrada

Notebook 2

Componente	Especificação principal	Detalhes
Processador (CPU)	11th Gen Intel(R) i5-11300H	4 cores, 8 threads. Frequência Base: 3.1GHZ
RAM	8Gb 3200MHZ	-

2.4 Procedimentos para medição de tempo

A medição de tempo foi realizada usando a função omp_get_wtime() da biblioteca OpenMP, que retorna o tempo em segundos com precisão de microssegundos.

2.5 Métricas utilizadas para avaliação

Selecionamos três métricas principais para a avaliação: tempo de execução, speedup (temp_par / temp_seq) e eficiência (speedup / num_threads). Essas foram calculadas para cada tamanho N e número de threads.

3. RESULTADOS

3.1 Tabelas e gráficos

Foram utilizados os parâmetros: tempo, speedup e eficiência em comparação com tamanho da matriz em cada teste.

tamMatriz	tempoSequencial	tempo2Thread	tempo4Thread	tempo8Thread
128	0.000212	0.000644	0.000588	0.020172
256	0.001704	0.001351	0.001716	0.017425
512	0.013723	0.014095	0.008459	0.022718
1024	0.118134	0.083916	0.051203	0.038615
2048	1.771400	1.251890	0.726579	0.521861
4096	9.631800	7.313070	6.466470	3.706600

tamMatriz	speedup2Thread	speedup4Thread	speedup8Thread
128	0.329617	0.360838	0.010521
256	1.261980	0.993211	0.097816
512	0.973596	1.622220	0.604055
1024	1.407780	2.307150	3.059320
2048	1.414980	2.438000	3.394390
4096	1.317070	1.489500	2.598560

tamMatriz	eficiencia2Thread	eficiencia4Thread	eficiencia8Thread
128	0.164808	0.090209	0.001315
256	0.630989	0.248303	0.012227
512	0.486798	0.405555	0.075507
1024	0.703889	0.576788	0.382416
2048	0.707492	0.609501	0.424299
4096	0.658533	0.372375	0.324819

Após tabular os dados de saída do algoritmo, estruturamos gráficos para inferimos conclusões acerca do teste.

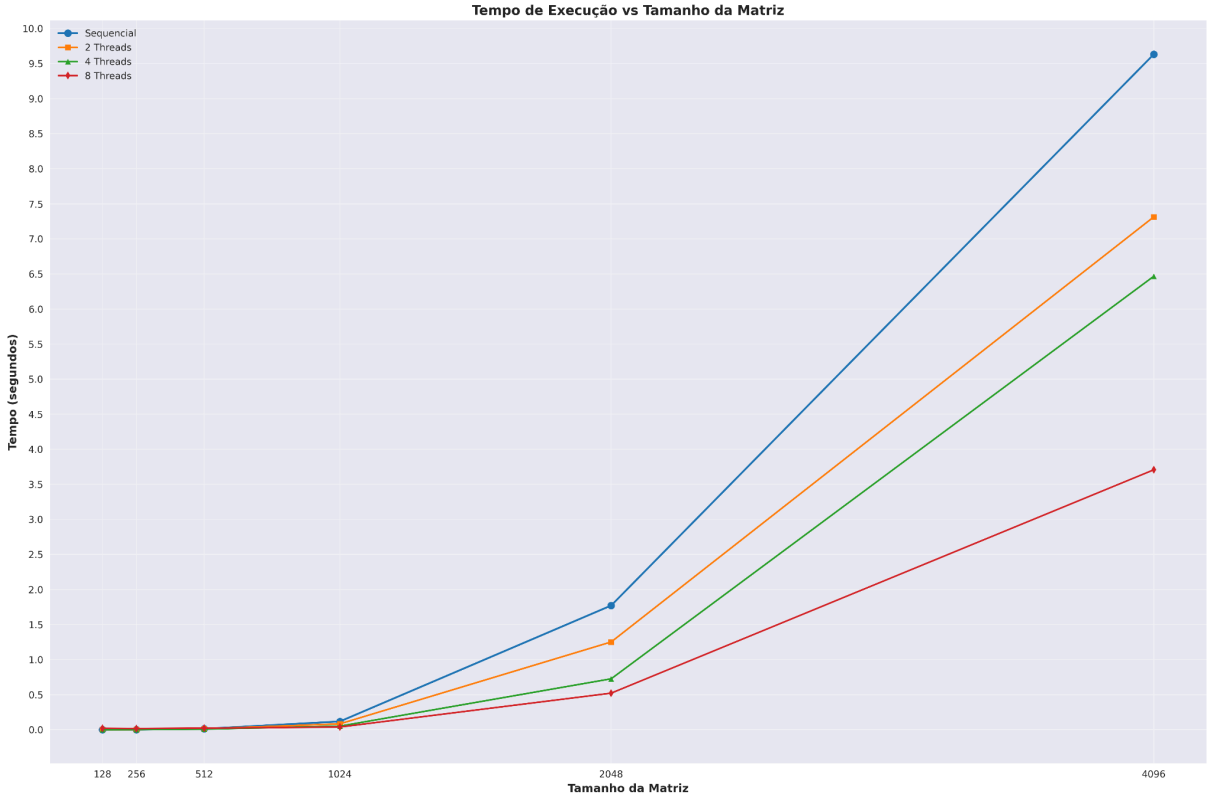

gráfico 1: Tempo de execução vs Tamanho da matriz

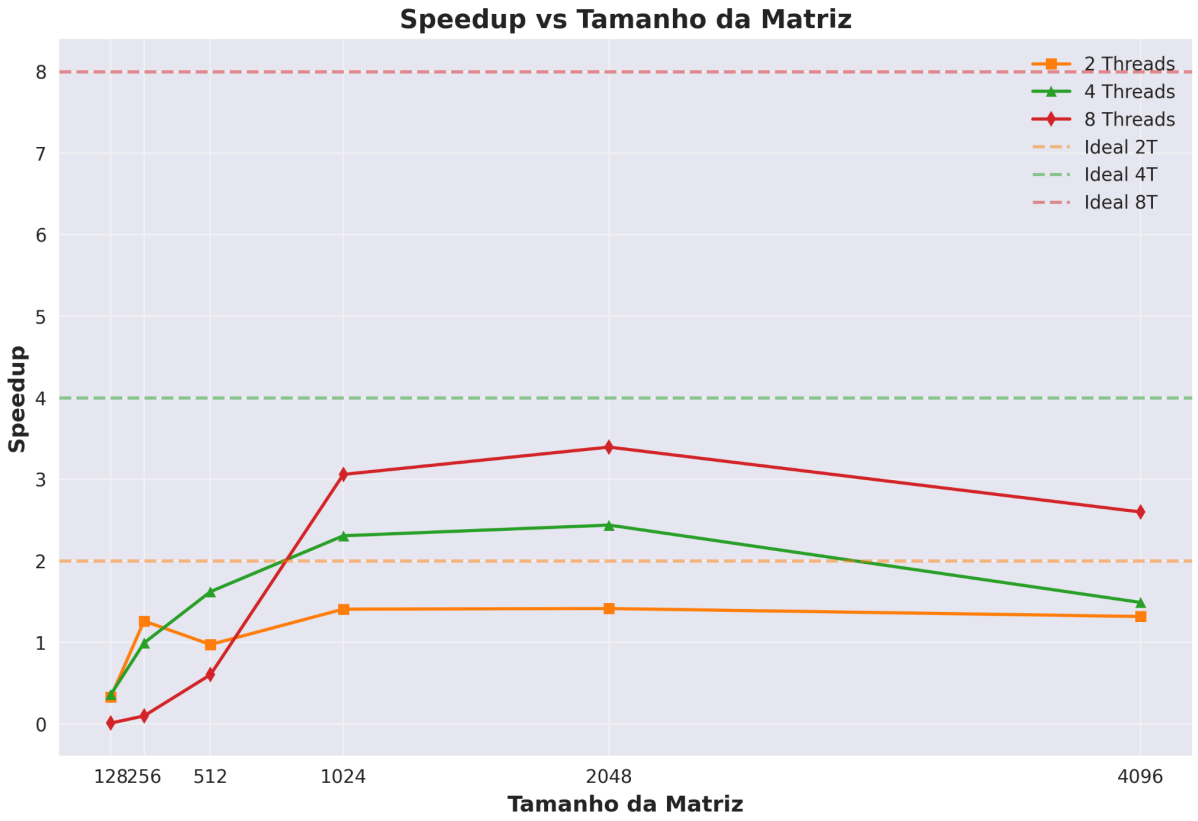

gráfico 2: Speedup vs Tamanho da matriz

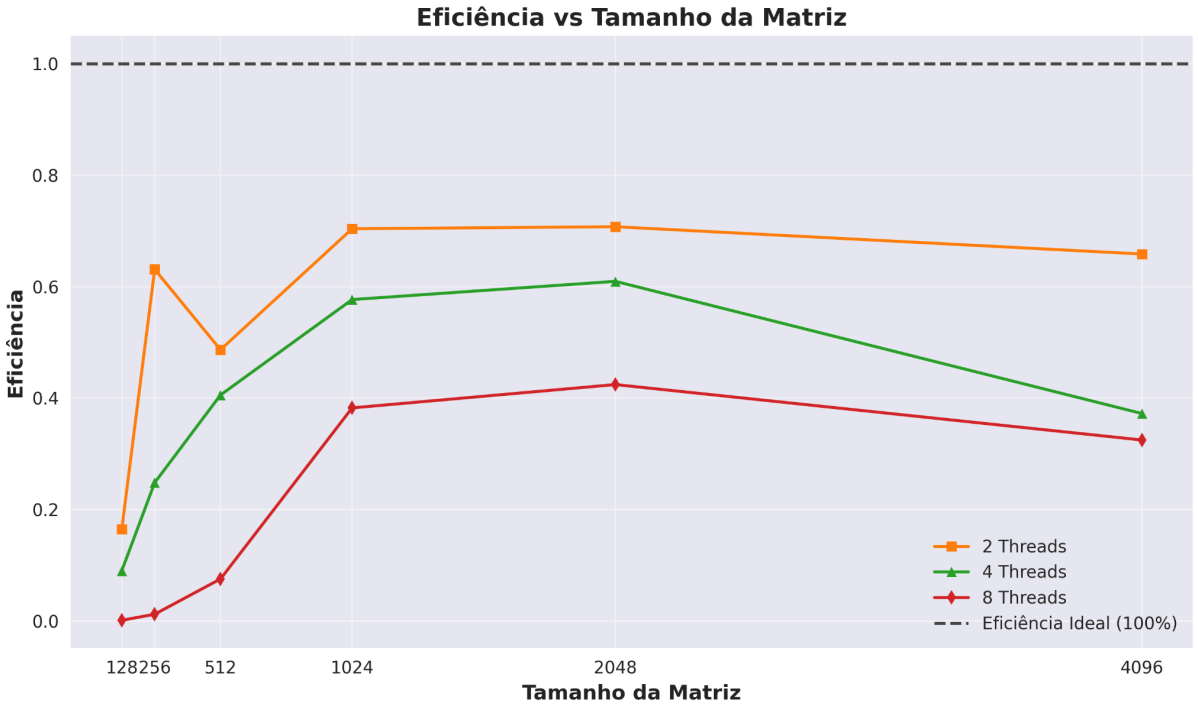

gráfico 3: Eficiência vs Tamanho da Matriz

3.2 Tempo, Speedup e eficiência

Em relação ao **tempo de execução**, observamos que o escalonamento do paralelismo apresenta retornos decrescentes. O tempo obtido com o número

máximo de threads testado (**8 threads**) mostra uma melhoria marginal em relação ao desempenho alcançado com a metade desse valor (**4 threads**).

Essa proximidade nos resultados de tempo se reflete diretamente no **Speedup**. O ganho de velocidade entre as configurações de 4 e 8 threads é **insuficiente para justificar a duplicação dos recursos**. Consequentemente, a métrica de **Eficiência** confirma essa limitação: o uso de 8 threads demonstra ser menos eficiente do que a configuração com 2 threads (um quarto do máximo) ou 4 threads, indicando que o sistema atingiu seu **ponto de saturação de paralelismo** devido, provavelmente, a gargalos de latência ou **contenção por recursos de memória (cache)**.

4. DISCUSSÃO

4.1 Sequencial Versus Paralela

A versão paralela, ao observar o pior caso no *gráfico 1* em relação ao algoritmo sequencial e o maior tamanho de matriz, concluí-se o ganho em tempo de execução do algoritmo com qualquer quantidade de threads, sendo provado pelo *gráfico 2* demonstrando o ganho de velocidade em relação ao sequencial (Speedup). Entretanto, esse ganho só cresce proporcionalmente ao aumento do tamanho das matrizes que estão sendo calculadas, perceptível a ineficácia da paralelização em tamanhos menores, com trabalhos mais leves. Ao criar, iniciar, e sincronizar os threads, este tempo gasto é maior que o tempo de cálculo, fazendo com que a versão sequencial, em tamanhos menores, se torne mais eficiente.

4.2 Impacto do aumento do número de threads

Como os dois notebooks a serem utilizados tinham quantidade de threads diferentes. Decidimos operar com até 8 threads. Para grande surpresa, o aumento de threads, não resultou num ganho pleno em relação ao custo de aumentar os recursos disponíveis para o algoritmo ser operado.

O ganho entre 4 threads e 8 threads é semelhante, apresentando retornos decrescentes. O speedup não é linear, logo, a adição de threads não é recompensada além do ponto de saturação. Conforme o *gráfico 3* o salto em ganho ocorreu entre 2 threads e 4 threads, cada um operando em seu bloco ideal. Bloco este que consideramos como comparação do sequencial, com 2 Threads, o ideal é que a operação seja 2x mais rápida que o sequencial, 4 threads 4x mais… e assim por diante.

Ademais, segundo a lei de amdahl, este ideal é inalcançável, em vista que uma porção do dgemm não pode ser paralelizada(leitura, inicialização etc.), o que sempre limita o ganho máximo. Por tanto, este limite ideal é somente amostral.

4.3 Sugestões de melhorias

O ideal seria utilizar otimizações de baixo nível, com verificação do gargalo na largura de banda/L3. Como usamos o tiling e ainda assim não conseguimos o resultado esperado, pensamos que até certo número de threads, eles competem para buscar dados que não cabem mais nos caches L1 e L2, sobrando para L3, tornando um fator limitante e dominante.

Ou, até mesmo o uso de outra estratégia para o uso de grande quantidades de threads conforme o tamanho das matrizes aumentam.

5. Considerações finais

O desenvolvimento deste projeto proporcionou um aprendizado significativo sobre os desafios e oportunidades do processamento paralelo na prática. A implementação do algoritmo DGEMM customizado permitiu não apenas exercitar conceitos teóricos vistos em aula, como a técnica de blocagem (tiling), o uso de diretivas OpenMP e a importância da localidade de acesso à memória, mas também compreender como esses elementos se relacionam para impactar o desempenho real de uma aplicação. Paralelizar um algoritmo vai além de simplesmente dividir o trabalho entre threads. A análise de speedup e eficiência mostrou que o ganho de desempenho tem limites práticos, muitas vezes definidos por gargalos de memória e overhead de sincronização, conforme previsto pela Lei de Amdahl. A experiência de testar diferentes números de threads e tamanhos de matriz reforçou a importância do dimensionamento adequado dos recursos, onde mais threads nem sempre significam melhor desempenho, especialmente quando o trabalho não é suficientemente grande para justificar o custo adicional de gerenciamento.

6. REFERÊNCIAS

https://terminalroot.com.br/2025/03/conheca-algumas-flags-para-melhorar-a-velocidade-do-seu-software.html
https://github.com/flame/how-to-optimize-gemm
https://github.com/etvorellana/RepoDEC107-PP-2025_2.git
https://open-catalog.codee.com/Glossary/Loop-tiling/
https://stackoverflow.com/questions/15829223/loop-tiling-blocking-for-large-dense-matrix-multiplication
https://pandas.pydata.org/docs/user_guide/index.html#user-guide
https://matplotlib.org/stable/index.html
Foram utilizados materiais disponíveis pelo professor no google classroom.