Projeto 1 — DGEMM Sequencial e Paralela com OpenMP

Autores:

Italo Santana Seara

Wilson Santos Silva Filho

Disciplina: DEC107 — Processamento Paralelo

Professor: Esbel Tomas Valero Orellana

Data: 28 de setembro de 2025

Conteúdo

1 Introdução

O projeto tem como objetivo explorar a computação paralela em arquiteturas de memória compartilhada, aplicando diretivas OpenMP para otimizar a multiplicação de matrizes de precisão dupla (DGEMM) e analisar os ganhos de desempenho em relação a uma implementação sequencial.

Como ponto inicial, o trabalho mergulha nos conceitos fundamentais do paralelismo, utilizando a poderosa ferramenta do OpenMP para acelerar uma das operações mais onipresentes na computação científica e na análise de dados: a multiplicação de matrizes. Ao desenvolver e comparar duas versões de uma rotina de multiplicação geral de matrizes de precisão dupla (DGEMM) (uma sequencial e outra paralela), teremos uma comparação utilizando métricas para identificar suas diferenças e eficiências.

A multiplicação de matrizes está presente em áreas como aprendizado de máquina, simulações físicas, bioinformática e processamento de imagens, onde a eficiência computacional pode impactar diretamente a viabilidade de soluções em grande escala. Dessa forma, ao explorar o uso de paralelismo com OpenMP, o trabalho não apenas contribui para o entendimento acadêmico do tema, mas também aproxima a pesquisa de cenários práticos onde o ganho de tempo de execução representa um diferencial estratégico.

A multiplicação geral de matrizes (GEMM) é uma operação fundamental em álgebra linear e computação científica. A variante DGEMM especifica que a operação é realizada com números em ponto flutuante de precisão dupla (double precision). A operação é definida como: a rotina `DGEMM` (Double-precision General Matrix Multiplication) implementa uma operação mais genérica do que uma simples multiplicação, dada por:

$$C = \alpha \cdot (A \cdot B) + \beta \cdot C$$

onde A, B e C são as matrizes, e α e β são escalares de precisão dupla.

Esta operação é um pilar da especificação BLAS (Basic Linear Algebra Subprograms), um conjunto de rotinas de baixo nível padronizadas para operações de álgebra linear. As implementações de BLAS, como a Intel MKL e OpenBLAS, são altamente otimizadas para arquiteturas de hardware específicas, explorando cache, vetorização e paralelismo para alcançar o máximo de desempenho. Neste projeto, a BLAS servirá apenas como uma referência de validação e comparação, não será chamada diretamente no código implementado.

A computação paralela é um paradigma que divide um problema computacional em partes menores que podem ser executadas simultaneamente por múltiplos processadores (ou núcleos). Em sistemas de memória compartilhada, todas as unidades de processamento (threads) têm acesso a um espaço de endereçamento de memória comum, o que simplifica a comunicação e o compartilhamento de dados entre elas. O principal desafio neste modelo é gerenciar o acesso concorrente a dados compartilhados para evitar con-

dições de corrida, onde o resultado da computação se torna incorreto devido à ordem imprevisível de execução das threads.

OpenMP é uma API padrão para programação paralela em memória compartilhada. Ela utiliza um modelo de execução fork-join, onde o programa inicia com uma única thread (a master thread). Ao encontrar uma região paralela, a master thread bifurca-se (fork), criando uma equipe de threads trabalhadoras que executam o código em paralelo. Ao final da região, as threads sincronizam e terminam, e apenas a master thread continua a execução (join).

O OpenMP simplifica a paralelização de código sequencial existente através de diretivas de compilador (pragmas), rotinas de biblioteca e variáveis de ambiente, permitindo ao desenvolvedor especificar quais partes do código devem ser paralelizadas, com controle sobre a distribuição de trabalho e a sincronização.

A simples paralelização de um código não garante um ganho de velocidade. Fatores como a sobrecarga de criação e gerenciamento de threads, a necessidade de sincronização e o desbalanceamento de carga podem degradar o desempenho, tornando a versão paralela mais lenta que a sequencial. Portanto, a análise de desempenho é crucial para validar a eficácia da paralelização. Ela permite quantificar os ganhos, entender os gargalos de escalabilidade e tomar decisões informadas sobre as estratégias de otimização. Em computação de alto desempenho (HPC), essa análise é fundamental para garantir o uso eficiente de recursos computacionais caros e para resolver problemas complexos em tempo hábil.

2 Metodologia

Nesta seção, detalharemos a abordagem adotada para implementar e avaliar as versões sequencial e paralela da multiplicação de matrizes DGEMM. Descreveremos os detalhes da implementação, o ambiente de teste, as métricas utilizadas para a avaliação de desempenho e o procedimento experimental.

2.1 Primeira implementação: versão sequencial

Inicialmente, uma implementação sequencial da multiplicação de matrizes foi o clássico algoritmo de três loops aninhados. A função `dgemm_sequencial` recebe como parâmetros os ponteiros para as matrizes A, B e C, bem como suas dimensões n, k e m. A matriz A tem dimensões $n \times k$, a matriz B tem dimensões $k \times m$, e a matriz C tem dimensões nm. A função calcula o produto das matrizes A e B, armazenando o resultado na matriz C. O código da função é apresentado a seguir:

```
1   void dgemm_sequencial(const double* A, const double* B,
2                         double* C, int n, int k, int m) {
3           for (int i = 0; i < n; ++i) {
4               for (int j = 0; j < m; ++j) {
5                   double sum = 0.0;
6                   for (int p = 0; p < k; ++p) {
7                       sum += A[i * k + p] * B[p * m + j];
8                   }
9                   C[i * m + j] = sum;
10              }
11          }
12      }
```

Cada elemento C_{ij} da matriz C é calculado pela soma dos produtos dos elementos correspondentes da i-ésima linha de A e da j-ésima coluna de B. Isso é expresso pela seguinte somatória:

$$C_{ij} = \sum_{p=0}^{k-1} A_{ip} \cdot B_{pj}$$

onde:

- i é o índice da linha, variando de 0 a $n - 1$.
- j é o índice da coluna, variando de 0 a $m - 1$.
- p é o índice da somatória, variando de 0 a $k - 1$.

2.2 Segunda implementação: versão paralela com OpenMP

A versão paralela da multiplicação de matrizes foi implementada utilizando diretivas OpenMP para paralelizar os loops externos da função `dgemm_paralela`. A diretiva `#pragma omp parallel` cria uma região paralela onde múltiplas threads são criadas para executar o código dentro do bloco. A diretiva `#pragma omp for` distribui as iterações do loop entre as threads disponíveis. A diretiva `#pragma omp barrier` garante que todas as threads tenham concluído suas iterações antes de prosseguir, evitando condições de corrida. O código da função paralela é apresentado a seguir:

```
1  void dgemm_paralela(const double* A, const double* B,
2            double* C, int n, int k, int m) {
3    #pragma omp parallel
4    {
5      #pragma omp for
6      for (int i = 0; i < n; ++i) {
7        for (int j = 0; j < m; ++j) {
8          double sum = 0.0;
9          for (int p = 0; p < k; ++p) {
10           sum += A[i * k + p] * B[p * m + j];
11         }
12         C[i * m + j] = sum;
13       }
14     }
15
16     #pragma omp barrier
17   }
18 }
```

Após a implementação, foram realizados testes para garantir que ambas as versões (sequencial e paralela) produzem resultados corretos e idênticos para as mesmas entradas. A validação foi feita comparando os elementos das matrizes resultantes C de ambas as versões, garantindo que a diferença entre os elementos correspondentes estivesse dentro de uma tolerância aceitável para números de ponto flutuante.

2.3 Otimizações

Após a implementação inicial, foram exploradas otimizações para melhorar o desempenho da versão paralela. As otimizações incluíram:

- **Transposição de matriz:** Transpor a matriz B para melhorar a localidade de referência durante a multiplicação, reduzindo falhas de cache.

- **Blocking (tiling):** Dividir as matrizes em blocos menores para melhorar a localidade de dados e reduzir o número de acessos à memória.
- **Vetorização:** Utilizar instruções SIMD (Single Instruction, Multiple Data) para otimizar a operação de multiplicação e soma dentro do loop mais interno.

Foram usadas as diretivas `#pragma omp parallel for` para paralelizar os loops de transposição e inicialização da matriz C. Na multiplicação, utilizou-se `#pragma omp parallel` com `#pragma omp for collapse(2) schedule(static)` para distribuir as iterações dos dois loops externos entre as threads, combinando ambos em um único loop e dividindo o trabalho de forma estática. No loop mais interno, aplicou-se `#pragma omp simd reduction(+:sum)` para vetorização e redução da variável `sum`, permitindo somas simultâneas e maior desempenho.

Com isso, a versão paralela otimizada foi implementada e testada novamente para garantir a correção dos resultados. Também fizemos uma versão sequencial otimizada para comparação.

A versão otimizada, paralela e sequencial, do código pode ser encontrada na pasta do projeto no arquivo `dgemm.c` junto com as outras implementações

2.4 Descrição do hardware utilizado nos testes

Os testes foram realizados em uma máquina com as seguintes especificações:

- **Processador:** Ryzen 7 5700G (8 núcleos, 16 threads)
- **Memória RAM:** 32 GB
- **Sistema Operacional:** Windows 11 + WSL 2.4.10.0 (Ubuntu 24.04.2 LTS)
- **Compilador:** GCC 13.3.0

2.5 Métricas utilizadas para avaliação

Usaremos as seguintes métricas para definir e comparar as implementações:

2.5.1 Tempo de Execução

Medido em segundos, é o tempo total gasto para completar a multiplicação das matrizes. Não inclui o tempo de inicialização ou finalização do programa, apenas o tempo gasto na execução da função de multiplicação.

$$T = T_{end} - T_{start}$$

Onde:

- T é o tempo de execução.
- T_{start} é o tempo registrado no início da execução da função.
- T_{end} é o tempo registrado ao final da execução da função.

Esse procedimento é feito com a função `omp_get_wtime()` do OpenMP, que retorna o tempo em segundos desde uma época fixa.

Foi utilizado um script em python para automatizar a execução dos testes, coletar os tempos e calcular as métricas de speedup e eficiência. O script cria matrizes de tamanhos variados, que são passadas como entrada para ambas as versões do código (sequencial e paralela), executa cada versão múltiplas vezes para obter uma média dos tempos, calcula as métricas de desempenho e gera um relatório com os resultados, que foram utilizados para criar os gráficos apresentados na seção de resultados.

2.5.2 Speedup

Mede o ganho de desempenho da versão paralela em relação à sequencial.

$$S_p = \frac{T_s}{T_p}$$

Onde:

- S_p é o speedup com p processadores/threads.
- T_s é o tempo de execução da versão sequencial.
- T_p é o tempo de execução da versão paralela com p processadores/threads.

2.5.3 Eficiência

Mede quão bem os recursos de processamento estão sendo utilizados pela versão paralela. Onde 1 é a eficiência ideal (100%).

$$E_p = \frac{S_p}{p} = \frac{T_s}{p \cdot T_p}$$

Onde:

- E_p é a eficiência com p processadores/threads. Varia entre 0 e 1.
- S_p é o speedup com p processadores/threads.
- p é o número de processadores/threads.
- T_s é o tempo de execução da versão sequencial.
- T_p é o tempo de execução da versão paralela com p processadores/threads.

3 Resultados

3.1 Sem otimizações

Inicialmente, os testes foram realizados com a versão paralela sem otimizações. Foram testados tamanhos de matrizes quadradas de 512 até 4096 (incrementando de 512 em 512) e números de threads variando de 1 a 16 (em potências de 2).

3.1.1 Tempo de execução

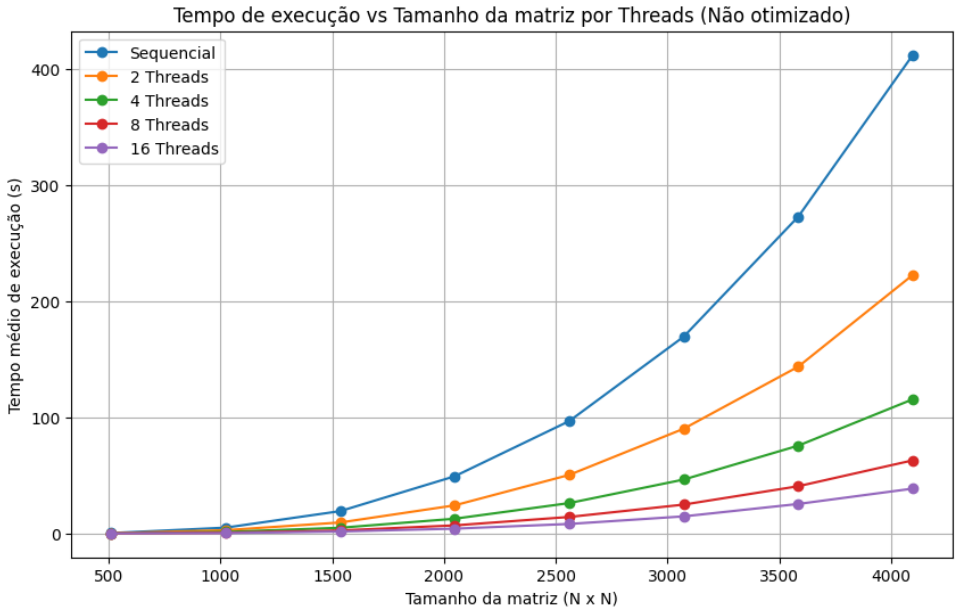

Figura 1: Tempo de execução (sem otimizações)

A Figura 1 mostra o tempo de execução das versões sequencial e paralela (sem otimizações) para diferentes tamanhos de matrizes e números de threads. Observa-se que a versão paralela tem um desempenho significativamente melhor, especialmente para matrizes maiores e com mais threads. A relação entre o tempo de execução e o número de threads é notavel, tendo em vista que quando dobram-se o número de threads, o tempo de execução quase reduz pela metade.

3.1.2 Speedup

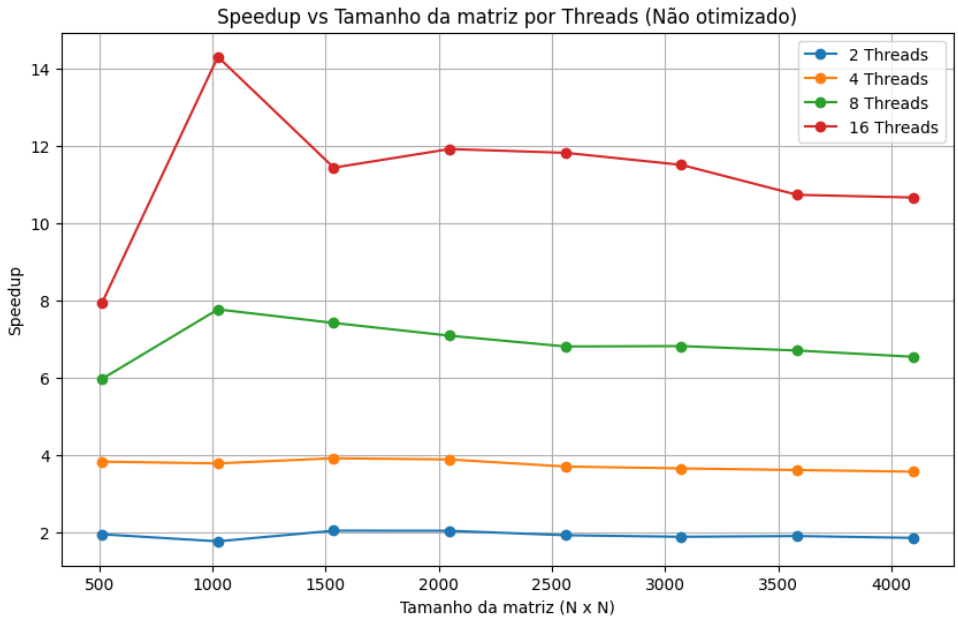

Figura 2: Speedup (sem otimizações)

A Figura 2 apresenta o speedup alcançado pela versão paralela em relação à sequencial para diferentes tamanhos de matrizes e números de threads. O speedup se mantém quase constante em relação ao tamanho da matriz, o que já era esperado, mas aumenta quase linearmente com o número de threads, indicando uma boa escalabilidade da implementação paralela.

3.1.3 Eficiência

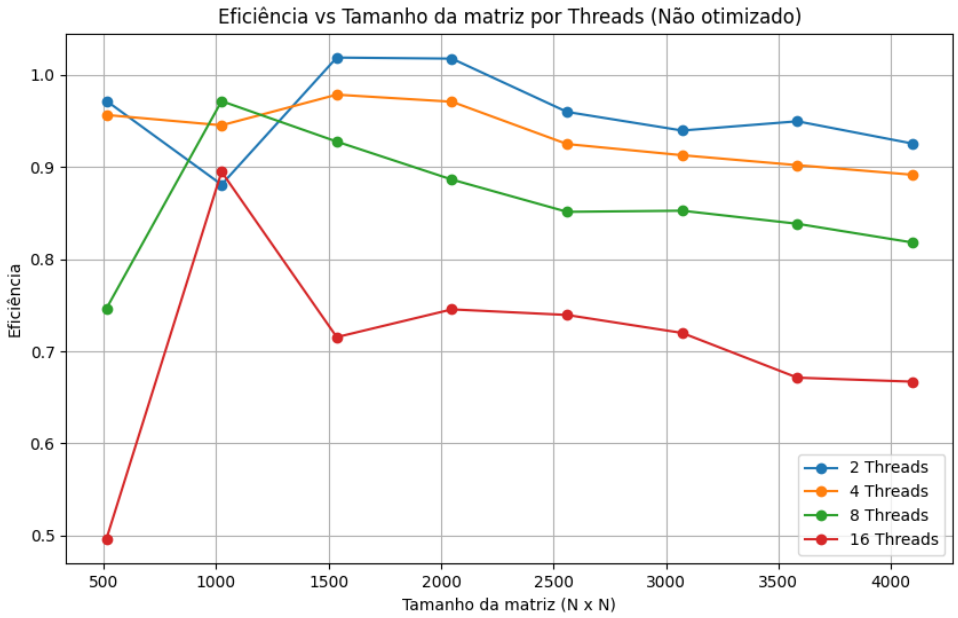

Figura 3: Eficiência (sem otimizações)

A Figura 3 mostra a eficiência da versão paralela em relação ao número de threads e tamanhos de matrizes. Observa-se que a eficiência diminui com o aumento do número de threads, o que é esperado devido à sobrecarga de gerenciamento de threads. No entanto, a eficiência se mantém relativamente alta (acima de 0.8) para até 8 threads, indicando que a paralelização é eficaz.

3.2 Com otimizações

Após implementar as otimizações, os testes foram repetidos com a versão paralela otimizada. Foram testados os mesmos parametros utilizados anteriormente.

3.2.1 Tempo de execução

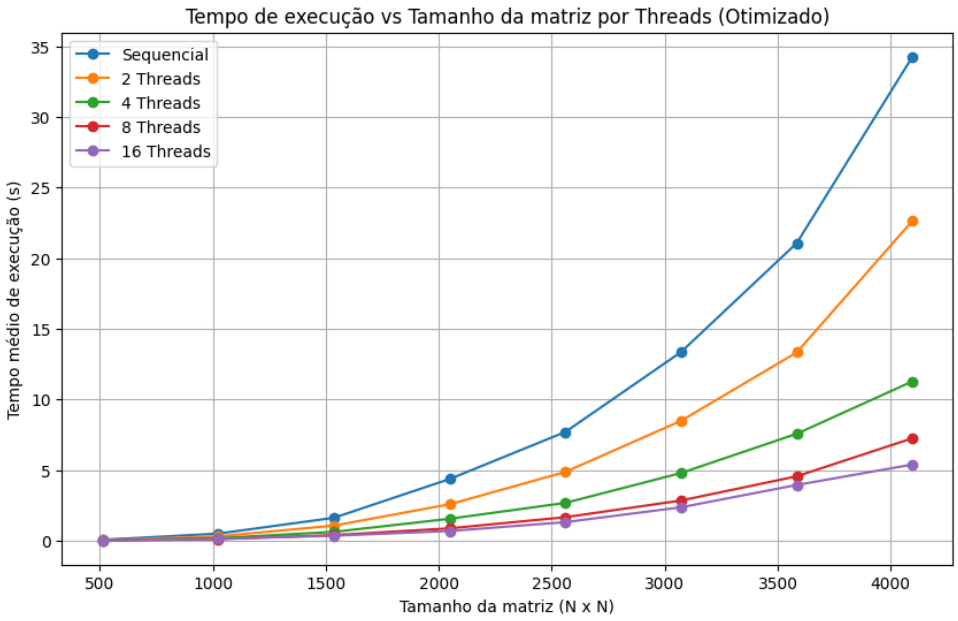

Figura 4: Tempo de execução (com otimizações)

É possível observar na Figura 4 que o tempo de execução da versão otimizada é significativamente menor do que o da versão sem otimizações, especialmente para matrizes maiores. Fazendo com que o tempo de execução caia de mais de 400 segundos para menos de 35 segundos no o pior caso em ambas as versões. O tempo de execução diminui ainda mais com o aumento do número de threads, chegando até a cerca de 5 segundos para 16 threads em seu pior caso.

3.2.2 Speedup

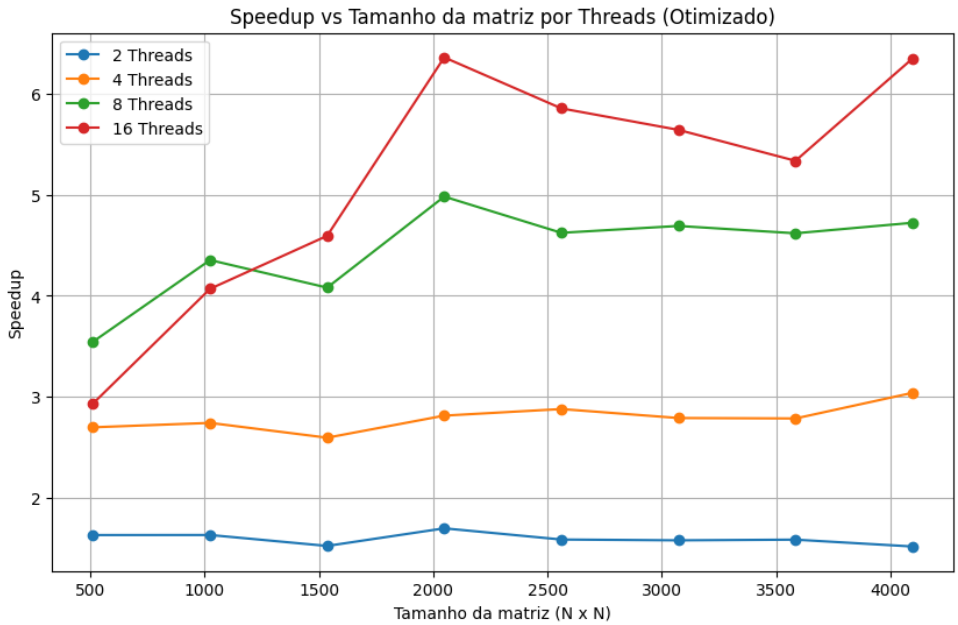

Figura 5: Speedup (com otimizações)

Na Figura 5, o speedup da versão otimizada é visivelmente menor em relação à versão sem otimizações. Isso se deve ao fato de que a versão sequencial otimizada também teve uma melhora significativa, reduzindo o ganho relativo da versão paralela. Também é possível observer um overhead computacional para os problemas menores, visto que o speedup para 16 threads é menor do que para 8 threads em matrizes de tamanho 512 e 1024. No entanto, o speedup ainda aumenta com o número de threads, para problemas maiores, indicando que a paralelização continua sendo eficaz.

3.2.3 Eficiência

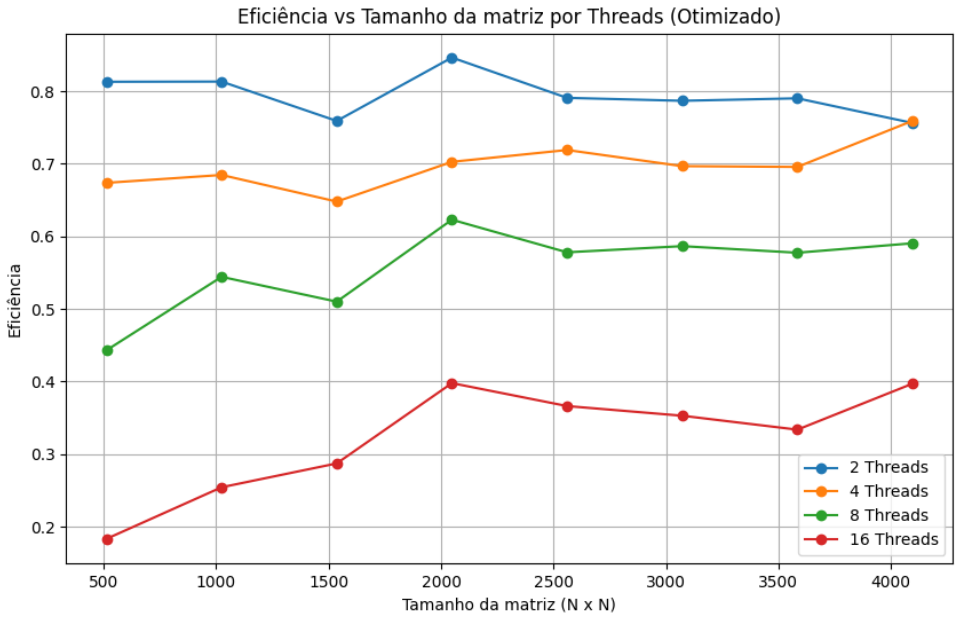

Figura 6: Eficiência (com otimizações)

A Figura 6 mostra que a eficiência da versão otimizada também possui uma eficiência menor em relação à versão sem otimizações. Isso é esperado, pois o speedup diminuiu. No entanto, a eficiência se mantém acima de 0.5 para até 8 threads, o que ainda é um resultado positivo, considerando as otimizações aplicadas.

4 Discussão

4.1 Comparação entre as versões sequencial e paralela

A versão paralela simples (cada thread calcula um subconjunto de linhas de `C`) reduz o tempo de execução relativo à implementação sequencial por explorar paralelismo de dados nas linhas da matriz resultante. Contudo, a versão paralela otimizada (transposição de `B` + tiling) apresenta desempenho substancialmente melhor do que as versões não otimizadas, a transposição melhora a localidade de acesso em `B`, o tiling reduz faltas de cache e as diretivas OpenMP bem aplicadas minimizam overhead de paralelismo e maximizam uso de recursos. A versão sequencial otimizada também é significativamente mais rápida que a paralela simples com 16 threads, mostrando que otimizações de memória e localidade são cruciais mesmo sem paralelismo. Em termos práticos, a ordem de desempenho observada costuma ser: `dgemm_parallel_opt` > `dgemm_sequential_opt` > `dgemm_parallel` > `dgemm_sequencial`, refletindo a importância tanto do paralelismo quanto da otimização de memória.

4.2 Impacto do aumento do número de threads

O ganho com aumento de threads não é linear. Inicialmente há aceleração significativa, mas com o aumento das threads surgem retornos decrescentes por causa de overhead de criação/sincronização, e possíveis efeitos de acesso de memória em sistemas com múltiplos sockets. Além disso, para tamanhos de problema pequenos o overhead de paralelização pode anular o benefício. A eficiência tende a cair com mais threads, pois o speedup não cresce na mesma proporção. Isso é típico em paralelismo devido a fatores como contenção de memória, overhead de gerenciamento de threads e desequilíbrio de carga. Em geral, há um ponto ótimo de threads para cada tamanho de problema, onde o ganho máximo é alcançado antes que o overhead comece a dominar.

4.3 Gargalos e limitações encontradas

Principais gargalos identificáveis a partir do código:

- **Banda de memória:** múltiplas threads acessando grandes matrizes podem saturar a memória principal, limitando o ganho.
- **Localidade de dados:** sem transposição e tiling, acessos a `B` são com saltos, causando muitas faltas de cache.
- **Overhead de paralelismo:** regiões paralelas muito curtas ou sincronizações frequentes (barriers) reduzem eficiência.
- **Alocação de `Bt`:** alocar e liberar `Bt` a cada chamada pode custar tempo.

- **Balanceamento de carga:** com `schedule(static)` por blocos, blocos finais menores podem causar leve desequilíbrio dependendo dos tamanhos.

4.4 Sugestões de melhorias

Para melhorar desempenho e escalabilidade sugiro, por ordem prática de impacto:

- Reusar a memória de `Bt` entre chamadas (evitar malloc/free repetidos) e garantir alinhamento adequado.
- Experimentos com diferentes tamanhos de bloco (`BS`) para ajustar ao conjunto de caches da máquina; considerar bloqueio adaptativo.
- Testar `schedule(dynamic, ...)` para casos com blocos de tamanhos irregulares ou usar heurísticas para balanceamento.
- Medir com ferramentas de profiling para identificar o verdadeiro gargalo antes de otimizações adicionais.

Em suma, as otimizações de memória (transposição, tiling) e a vetorização trazem ganhos tão ou mais importantes que a simples paralelização. Combinar afinamento do `BS`, reuso de memória e estratégias de escalonamento pode levar a melhorias adicionais significativas.

5 Conclusão

5.1 Principais aprendizados do projeto

Este projeto proporcionou uma compreensão prática e teórica dos conceitos de paralelismo em memória compartilhada. A implementação da multiplicação de matrizes DGEMM, tanto na versão sequencial quanto na paralela, permitiu explorar os desafios e benefícios da computação paralela. Além disso, a aplicação de otimizações como transposição de matrizes e tiling destacou a importância da localidade de dados e do acesso eficiente à memória para o desempenho computacional. A análise de desempenho, incluindo métricas como tempo de execução, speedup e eficiência, foi fundamental para avaliar a eficácia das implementações e entender os trade-offs envolvidos. Em resumo, o projeto reforçou a importância do paralelismo e das otimizações de memória na computação de alto desempenho.

5.2 Considerações sobre desempenho obtido

Comparando a implementação original (sequencial sem otimizações) com a versão final que foi implementada, para matrizes de tamanho 4096×4096, o tempo de execução caiu de aproximadamente 412 segundos para cerca de 5.4 segundos na versão paralela otimizada com 16 threads. Isso representa uma redução de tempo de execução em torno de 7,650%, evidenciando o impacto significativo das otimizações aplicadas. O speedup alcançado foi substancial, embora não linear, devido aos fatores discutidos anteriormente, como overhead de paralelismo e limitações de banda de memória. A eficiência da versão paralela otimizada se manteve relativamente alta, especialmente para até 8 threads, indicando que as estratégias adotadas foram eficazes em maximizar o uso dos recursos computacionais disponíveis.

Referências

[1] *Orellana, E.*, Materiais de slides vistos em aula

[2] *OPENMP*, Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-RefGuide-6.
Acesso em: 22 de Setembro de 2025.

[3] Brasil Escola, Disponível em: https://brasilescola.uol.com.br/matematica/multiplicacao-m
Acesso em: 22 de Setembro de 2025.

[4] VSP-BERLIN, Disponível em: https://svn.vsp.tu-berlin.de/repos/public-svn/publication
Acesso em: 23 de Setembro de 2025.

[5] Wikipedia, Disponível em: https://en.wikipedia.org/wiki/Loop_nest_optimization.
Acesso em: 23 de Setembro de 2025.