Projeto 3 — DGEMM Sequencial e Paralela para processamento em GPGPUs com CUDA

Autores:

Italo Santana Seara

Wilson Santos Silva Filho

Disciplina: DEC107 — Processamento Paralelo

Professor: Esbel Tomas Valero Orellana

Data: 27 de novembro de 2025

Conteúdo

1 Introdução

A multiplicação de matrizes é uma operação fundamental em diversas áreas da computação científica e engenharia, presente em aplicações que vão desde álgebra linear numérica até aprendizagem de máquina e simulações físicas. Neste projeto, focamos na versão de precisão dupla da multiplicação geral de matrizes (DGEMM), cujo problema pode ser formalizado como: dado $A \in \mathbb{R}^{m \times k}$ e $B \in \mathbb{R}^{k \times n}$, calcular $C = AB$, onde

$$C_{ij} = \sum_{t=1}^{k} A_{it} B_{tj}, \qquad 1 \leq i \leq m,\ 1 \leq j \leq n.$$

A complexidade aritmética desta operação cresce como $\mathcal{O}(mkn)$ e, para matrizes quadradas de ordem N, como $\mathcal{O}(N^3)$, o que torna essencial a exploração de paralelismo para resolver instâncias de grande porte de forma eficiente.

No Projeto 1 foram implementadas versões sequenciais e paralelas (por memória compartilhada, com OpenMP) da DGEMM. No Projeto 2, foi implementado a versão com o MPI, onde o estudo do paralelismo migrando o algorítimo para modelo de memória distribuída. No presente projeto 3, daremos continuidade ao estudo de processamento paralelo implementando a DGEMM em GPGPUs (General-Purpose computing on Graphics Processing Units) utilizando o alto poder de processamento desses *devices*.

Este projeto tem como objetivos específicos:

- Implementar uma versão paralela de DGEMM utilizando a programação de GPGPUs para Nvidia (CUDA) e utilizar algumas rotinas (por exemplo, `cudaMemoryCopy`, implementar estrutura hierárquica de memória (`__global__, __device__, __host__ e __shered__`,)).

- Comparar o resultado da versão CUDA com o resultado da versão sequencial, calcular a diferença relativa máxima.

- Medir e analisar desempenho (tempo total, *speedup* e eficiência) em várias configurações de blocos e threds e tamanhos de matrizes, e comparar os resultados com os obtidos em ambiente de memória compartilhada (OpenMP, MPI).

- Comparar os resultados com os projetos 1 e 2, apresentando discussões sobre o ganho obtido com o uso da GPU e a diferença de escalabilidade entre os modelos de paralelismo

2 Metodologia

Nesta seção, detalhamos a implementação da multiplicação de matrizes DGEMM em suas versões sequencial e paralela utilizando programação CUDA, bem como a metodologia adotada para medir desempenho e validar diferença relativa máxima.

2.1 Implementação Sequencial

A implementação sequencial da DGEMM utiliza a versão otimizada do Projeto 1, que emprega blocos para melhorar a localidade de dados. As matrizes são armazenadas em vetores unidimensionais (*row-major order*) para garantir um acesso eficiente à memória. A função principal responsável pela multiplicação é apresentada no repositório do projeto.[1]

Esta implementação serve como referência para comparação de desempenho com a versão paralela da GPU com CUDA.

2.2 Implementação Paralela com OpenMP

A versão paralela com OpenMP, desenvolvida no Projeto 1, utiliza diretivas de paralelismo para distribuir o trabalho entre múltiplas threads em um ambiente de memória compartilhada. A decomposição por blocos é mantida, e a paralelização é aplicada principalmente nos loops externos da multiplicação de matrizes.

Esta implementação também está disponível no repositório do projeto.[1]

2.3 Implementação Paralela com MPI

A versão paralela da DGEMM desenvolvida utilizando o padrão MPI no Projeto 2, que permite a comunicação entre processos em um ambiente de memória distribuída, utilizou-se de estratégias que envolve e adota a decomposição das matrizes por blocos, onde cada processo é responsável por calcular uma parte da matriz resultado C. Esta implementação também está disponível no repositório do projeto.

2.4 Implementação Paralela para GPGPUs com CUDA

A versão paralela da DGEMM foi desenvolvida utilizando a programação para GPGPUs da Nvidia, utilizando CUDA. A GPU possui milhares de núcleos, organizar quem faz o quê é crucial. As GPUS usam um sistema de coordenadas onde a Thread: A menor unidade de execução. Cada thread roda o Kernel. O Block (Bloco): Um grupo de threads que podem compartilhar memória rápida (Shared Memory) e se sincronizar e a Grid (Grade): O conjunto total de blocos que executam o kernel. A implementação completa está disponível no repositório do projeto.[1]

2.4.1 Decorators `__stricted__` e `__global__`

As funções de kernel CUDA são decoradas com o especificador `__global__`, indicando que são executadas na GPU e chamadas a partir do host (CPU). Além disso, o uso do decorador `__restrict__` nos ponteiros de entrada e saída informa ao compilador que esses ponteiros não se sobrepõem, permitindo otimizações adicionais durante a compilação.

2.4.2 Organização de Threads e Blocos

O código utiliza uma estratégia de decomposição de domínio em 2D, mapeando threads diretamente para as coordenadas da matriz de resultado C. Estrutura de Bloco (dim3 block): O kernel é configurado com blocos quadrados de tamanho tile x tile (por padrão, 32×32, totalizando 1024 threads por bloco, que é o máximo permitido na maioria das arquiteturas CUDA). Estrutura de Grade (dim3 grid): A grade é calculada para cobrir as dimensões da matriz C (M linhas por N colunas). O número de blocos é arredondado para cima ((N + tile - 1) / tile) para garantir que matrizes cujas dimensões não sejam múltiplos de 32 sejam processadas corretamente. Mapeamento: threadIdx.x mapeia para a coluna dentro do bloco (e do tile).threadIdx.y mapeia para a linha dentro do bloco.A coordenada global (row, col) é calculada como:

$$row = blockIdx.y \times blockDim.y + threadIdx.y$$

$$col = blockIdx.x \times blockDim.x + threadIdx.x$$

2.4.3 Uso de Memória Compartilhada `__shared__` e Hierarquia de Memória

A diferença crítica entre as funções `dgemm_kernel_basic` e `dgemm_kernel` é o uso da hierarquia de memória para contornar o gargalo de largura de banda da memória global (DRAM). A Otimização via Tiling: O `dgemm_kernel` implementa a técnica de Tiling (ladrilhamento). À Declaração: `extern __shared__ double smem[];` aloca memória rápida on-chip (L1/Shared), acessível por todas as threads do mesmo bloco. Utilizamos o uso de carregamento cooperativo: O código divide as matrizes A e B em sub-blocos (tiles) quadrados de tamanho $T \times T$. Todas as threads do bloco colaboram para carregar um tile de A e um tile de B da memória global (lenta) para a memória compartilhada (rápida). O comando `__`syncthreads() garante que todo o tile esteja carregado na memória compartilhada antes que qualquer thread comece a calcular. Do Cálculo: Uma vez que os dados estão na smem, as threads calculam o produto escalar parcial lendo apenas da memória compartilhada.

Isso causa um impacto: Em vez de fazer $2 \times K$ leituras da memória global por thread (como no kernel básico), cada elemento é lido da memória global apenas uma vez por tile e reutilizado T vezes. Isso reduz o tráfego de memória global por um fator de T (neste

caso, 32 vezes menos acesso à RAM da GPU).

2.4.4 Sobreposição de Computação e Comunicação

Se analisarmos algumas funções como a `dgemm_paralell_cuda` podemos observar um fluxo:
`cudaMemcpy(..., cudaMemcpyHostToDevice)` onde realiza-se a Cópia H->D
dgemm_kernel <<< ... >>> (Execução do Kernel) `cudaMemcpy(..., cudaMemcpyDeviceToHost)` (Cópia D->H)
Percebemos que não há sobreposição de computação e comunicação pois: As chamadas `cudaMemcpy` por padrão são bloqueantes (ou seja, síncronas em relação ao host ou seria lizadas nos stream padrão) Então, a GPU fica ociosa durante a transferencia de dados via PCLe, e o PCLe fica ocioso durante o calculo no Kernel.
Para implementações futuras: Para matrizes muito grandes, o tempo de transferência pode ser significativo, o que deveria ser usado com alicerce e as funções de CUDA Streams e AsyncMemcpy para que a GPU processe um pedaço da matriz enquanto o outro esta sendo transferido.

2.5 Diferença relativa máxima

Vamos validar a versão sequencial comparando os resultados obtidos com a função ....
A implementação sequencial (rodando na CPU) serve como a referência (ground truth). Ela é, por definição, a maneira mais simples e verificável de calcular o resultado (neste caso, a multiplicação de matrizes).

A diferença relativa máxima entre as duas matrizes de resultado C_{seq} (sequencial) e C_{cuda} (CUDA) é calculada como:

$$\Delta = \max_{i,j} \frac{|C_{\text{seq}}(i,j) - C_{\text{cuda}}(i,j)|}{|C_{\text{seq}}(i,j)| + \epsilon} \tag{1}$$

onde $\epsilon = 10^{-12}$ evita divisões por zero.

Considerar corretas as implementações cuja diferença máxima seja menor que um limite aceitável (por exemplo, 10^{-8}).

2.6 Descrição do hardware utilizado nos testes

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

- **Processador:** Ryzen 7 5700G (8 núcleos)

[1]Link para as implementações: https://github.com/italoseara/DEC107/blob/main/Projeto3/src/dgemm_cuda.cu

- **GPU:** NVIDIA GeForce RTX 3060 (12GB VRAM, 3584 CUDA Cores, Arquitetura Ampere)
- **Memória RAM:** 32 GB
- **Sistema Operacional:** Windows 11 + WSL 2.4.10.0 (Ubuntu 24.04.2 LTS)
- **Compilador:** GCC 13.3.0
- **GPU:** NVIDIA GeForce RTX 4060 (Dedicated GPU memory 8GB, Shared GPU memory 16GB, CUDA Cores 3072, Architecture Ada Lovelace)

2.7 Métricas utilizadas para avaliação

Usaremos as seguintes métricas para definir e comparar as implementações:

2.7.1 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.

Foi utilizado um script em python para automatizar a execução dos testes e coletar os tempos. O script executa cada versão múltiplas vezes com diferentes entradas 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.

2.7.2 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.

2.7.3 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

Nesta seção, apresentamos os resultados obtidos a partir dos testes realizados nas implementações sequencial, paralela com CUDA da multiplicação de matrizes DGEMM. O resultado vai ser comparado com versões paralelizadas utilizando OpenMP e MPI Os resultados incluem a validação de corretude - Diferença relativa maxima, tabelas e gráficos de desempenho, bem como cálculos de *speedup* e eficiência.

3.1 Tempo de execução

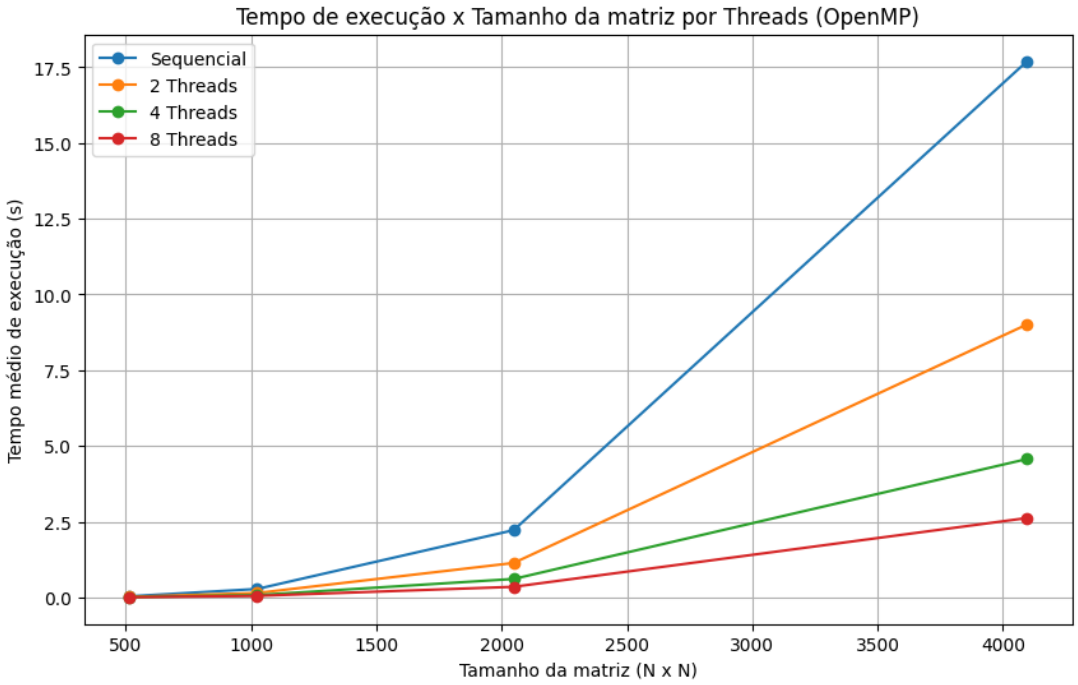

Figura 1: Tempo de execução (OpenMP)

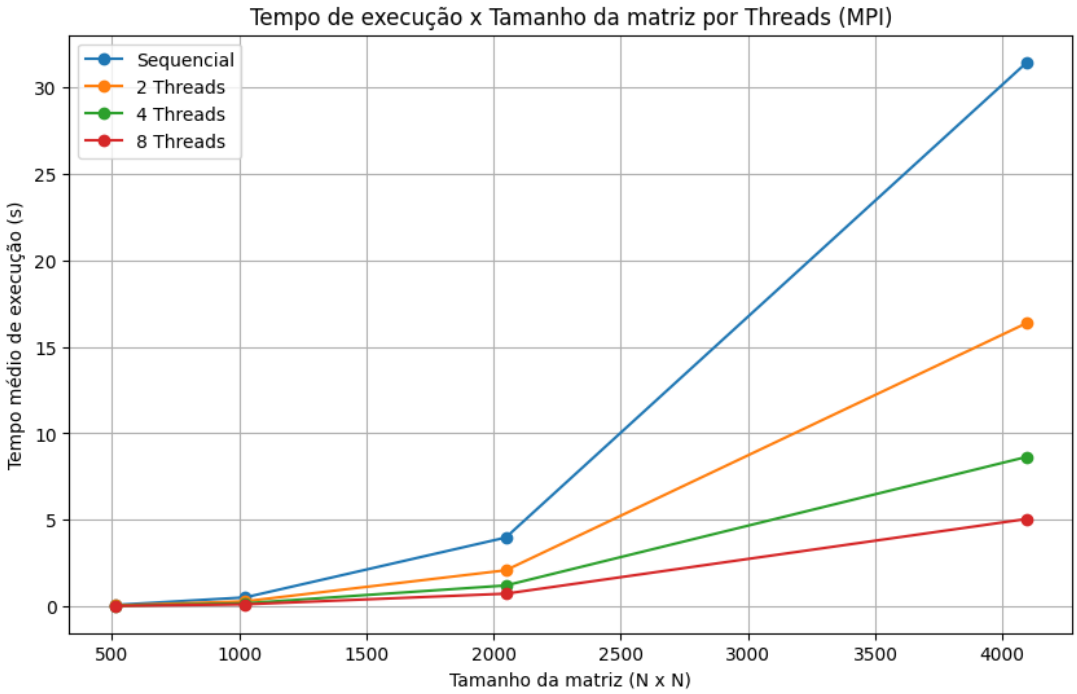

Figura 2: Tempo de execução (MPI)

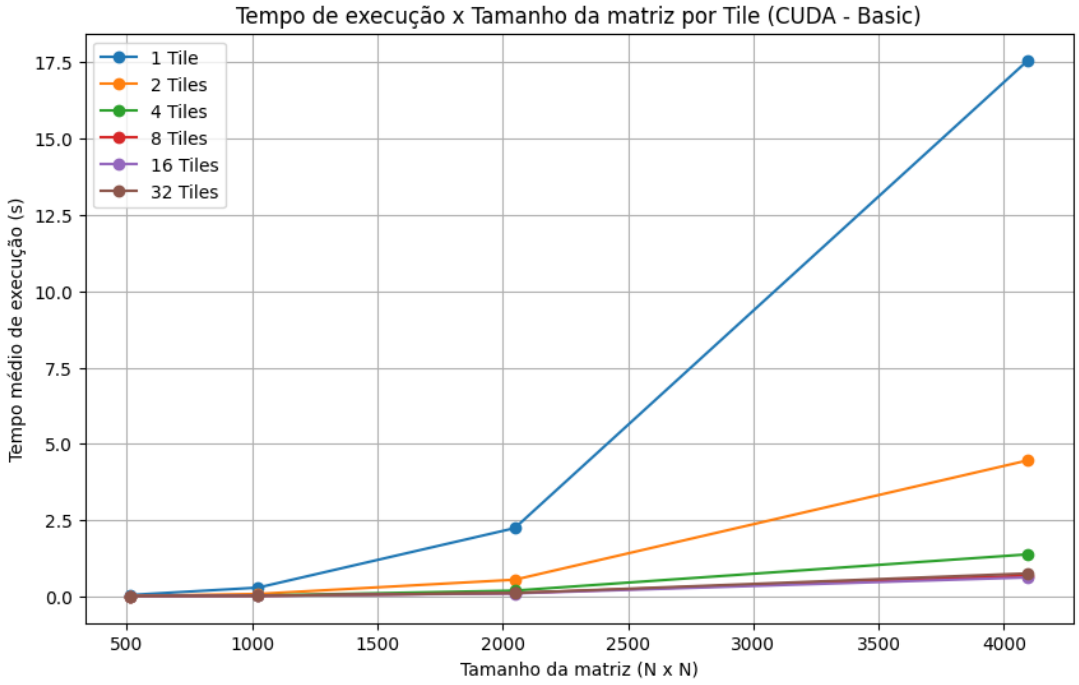

Figura 3: Tempo de execução (CUDA - Basic)

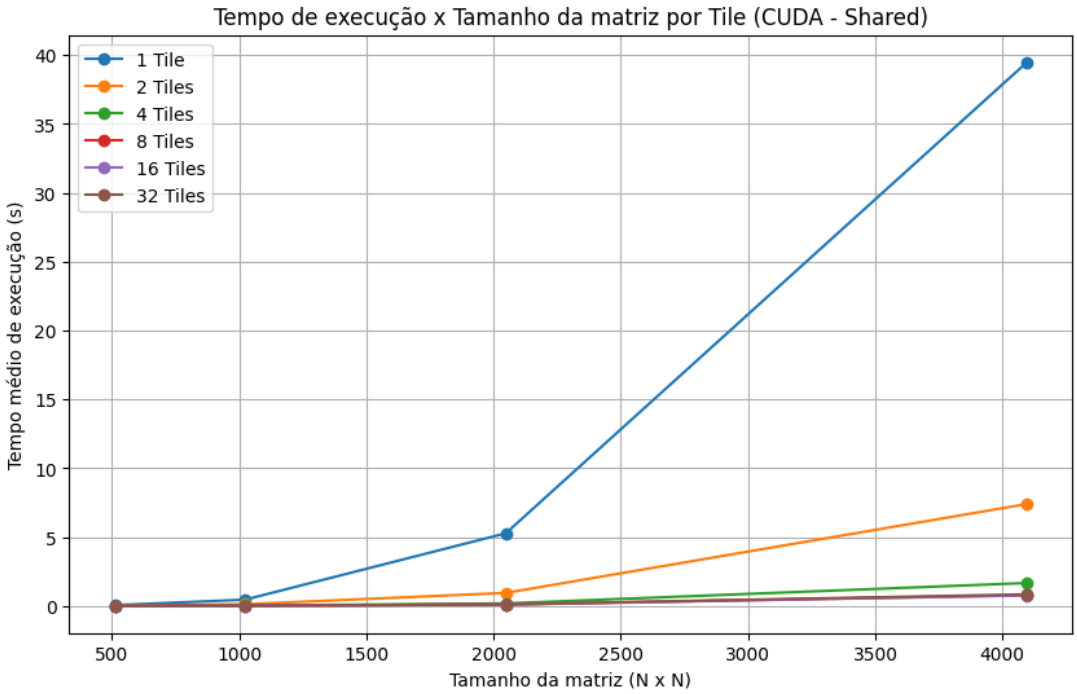

Figura 4: Tempo de execução (CUDA - Shared)

As Figuras 1 e 2 mostram o tempo de execução das implementações com OpenMP e MPI, respectivamente. Ambas as implementações apresentam uma redução significativa no tempo de execução com o aumento do número de threads/processos, evidenciando a eficácia do paralelismo. Podemos ver que a versao com OpenMP apresenta um desempenho ligeiramente melhor do que a versão com MPI, especialmente para matrizes maiores, o que pode ser atribuído à menor sobrecarga de comunicação no modelo de memória compartilhada utilizado pelo OpenMP. A Figura 3 mostra o tempo de execução da implementação básica com CUDA, que apresenta um desempenho significativamente melhor em comparação com as versões de CPU (OpenMP e MPI), especialmente para matrizes maiores. A Figura 4 mostra o tempo de execução da versão otimizada com memória compartilhada, o que é inesperado é que o tempo de execução da versão shared é significativo pior que a CUDA-Basic e openmp e MPI, isso se deve provavelmente ao uso da memoria cache L2 imensa da GPU (16MB).

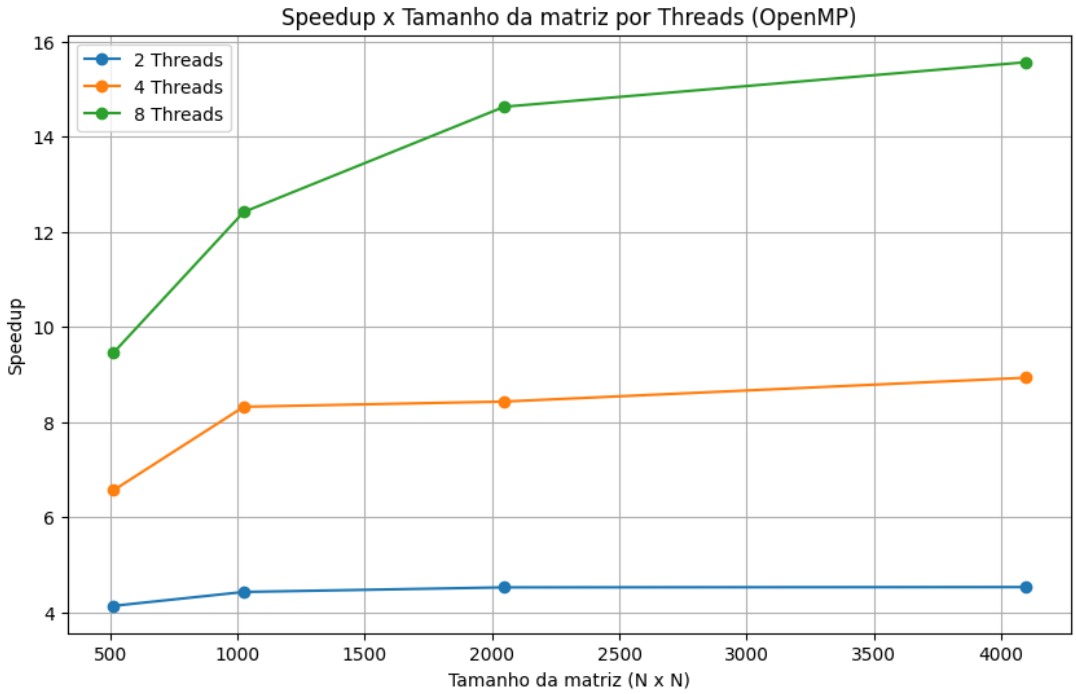

Figura 5: Speedup (OpenMP)

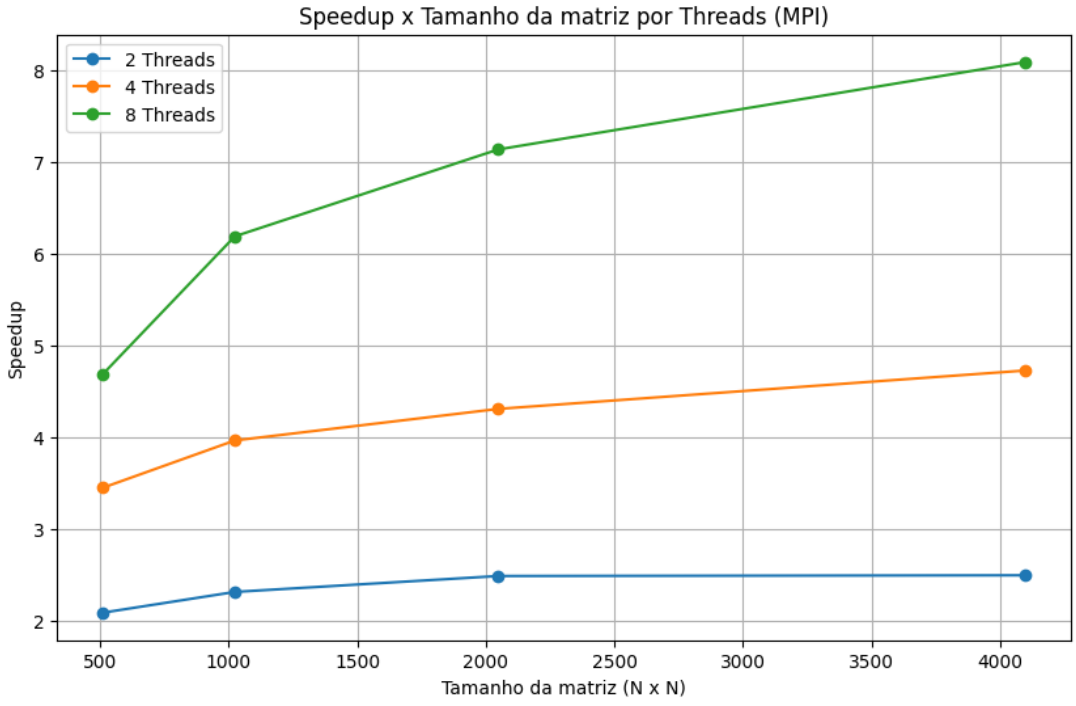

Figura 6: Speedup (MPI)

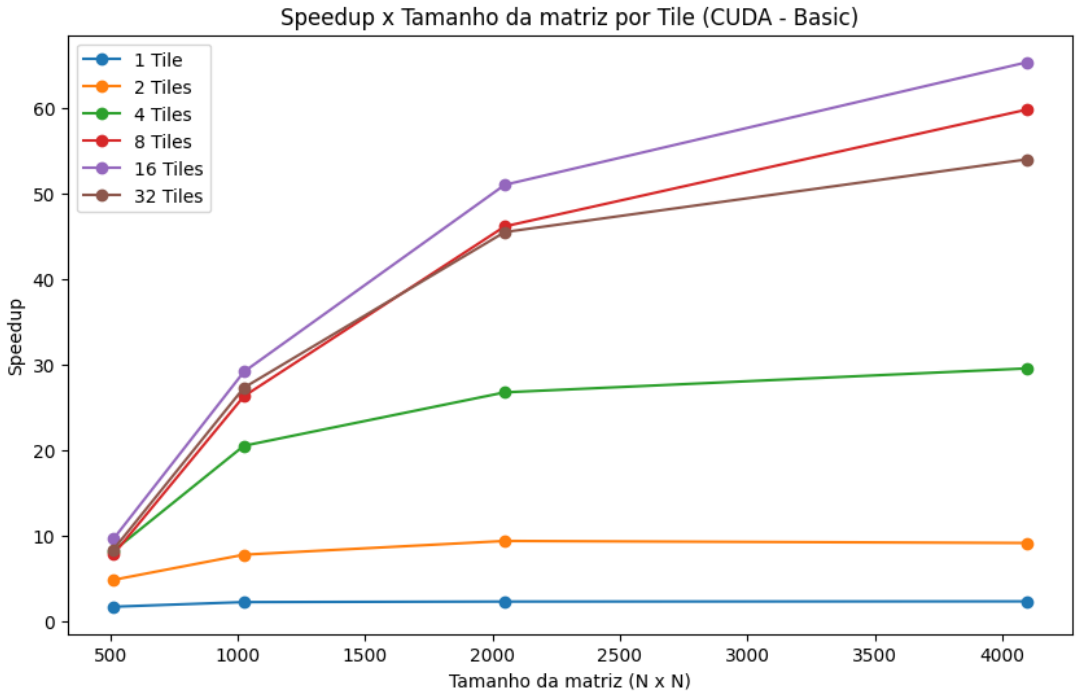

Figura 7: Speedup (CUDA - Basic)

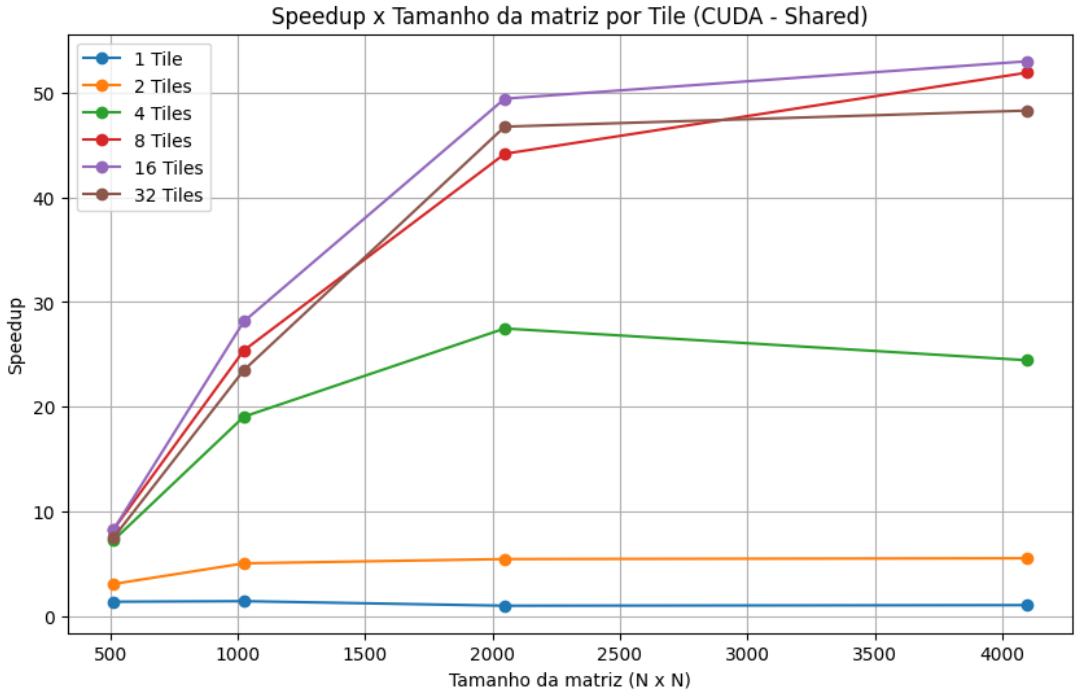

Figura 8: Speedup (CUDA - Shared)

As Figuras 5 e 6 mostram o speedup das implementações com OpenMP e MPI, respectivamente. Ambas as implementações apresentam um aumento no speedup com o aumento do número de threads/processos, embora o ganho diminua devido à sobrecarga de comunicação. A Figura 7 mostra o speedup da implementação básica com CUDA, que

apresenta um ganho significativo em relação à versão shared, especialmente para matrizes significativamente maiores, o que é esperado mas ao mesmo tempo estranho pois esperariamos o resultado contrário. A Figura 8 mostra o speedup da versão shared com memória compartilhada, que alcança um desempenho inferiror, mesmo destacando as eficiencias da GPU com o uso via tiling. Isso de deve ao espaço cache imenso da GPU utilizada (RTX 4060) que consegue armazenar grande parte das matrizes na cache L2 (16MB), reduzindo a necessidade de acesso à memória global mesmo na versão básica. Podemos perceber que a versão CUDA shared em muitos casos supera significativamente as implementações de CPU (OpenMP e MPI), evidenciando o poder do paralelismo massivo e das otimizações específicas para GPUs.

3.2 Eficiência

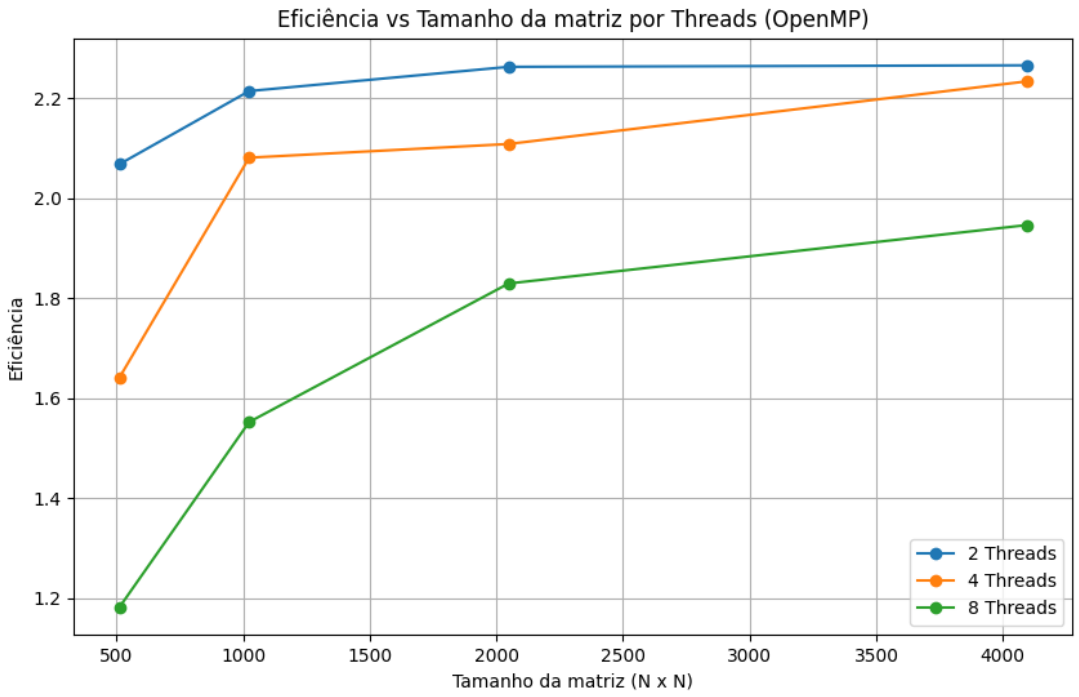

Figura 9: Eficiência (OpenMP)

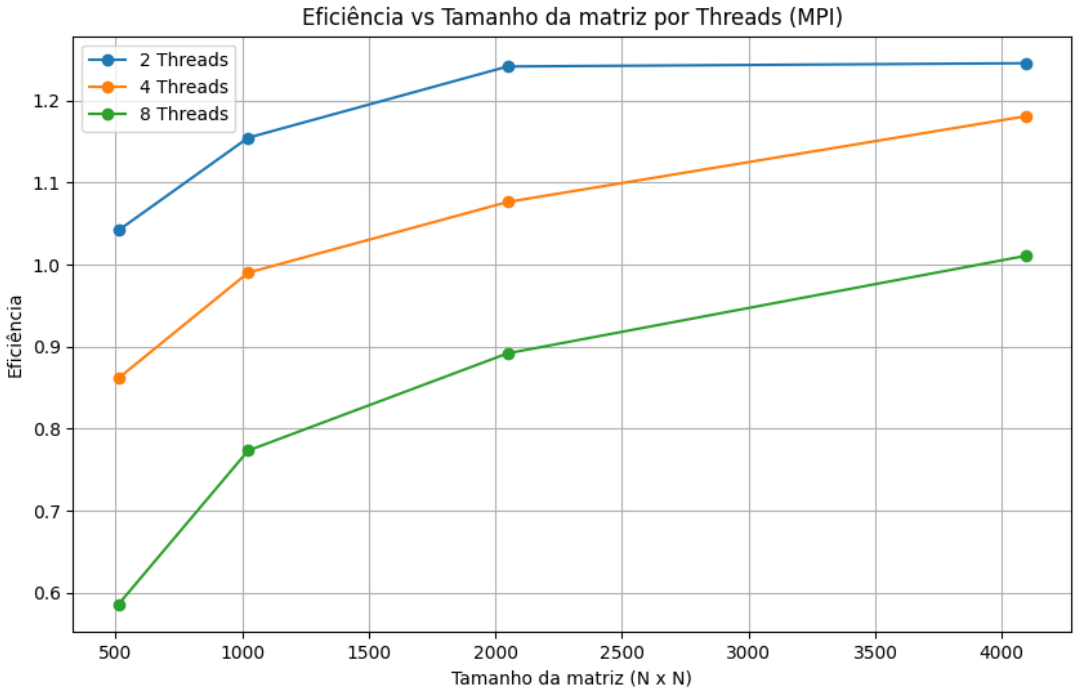

Figura 10: Eficiência (MPI)

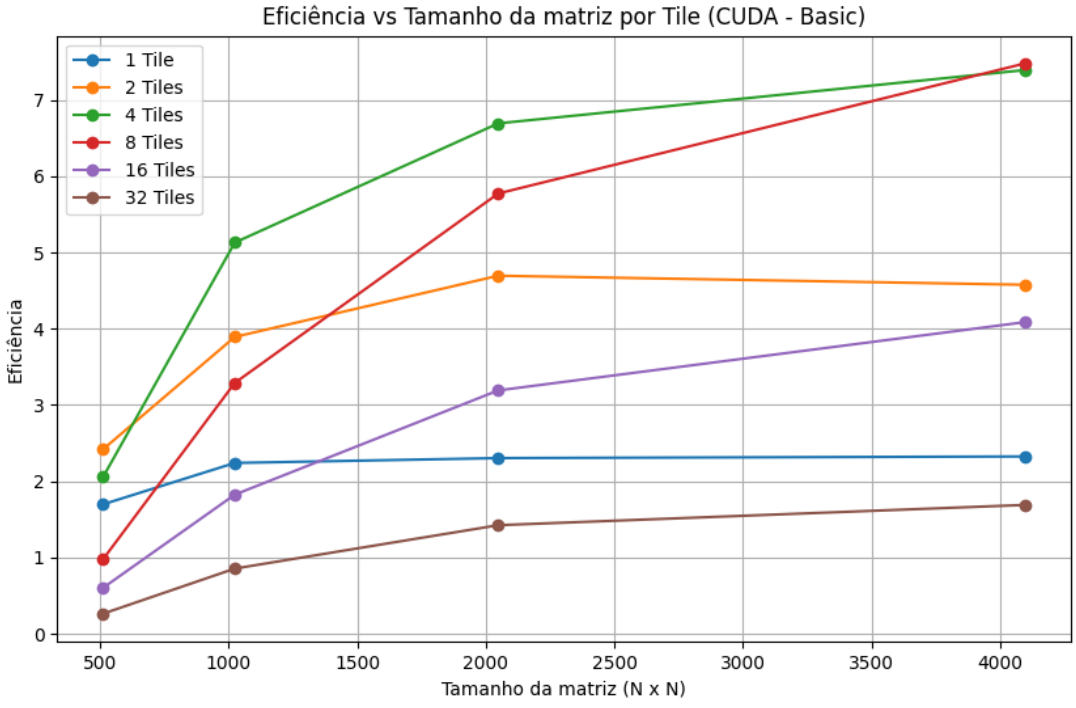

Figura 11: Eficiência (CUDA - Basic)

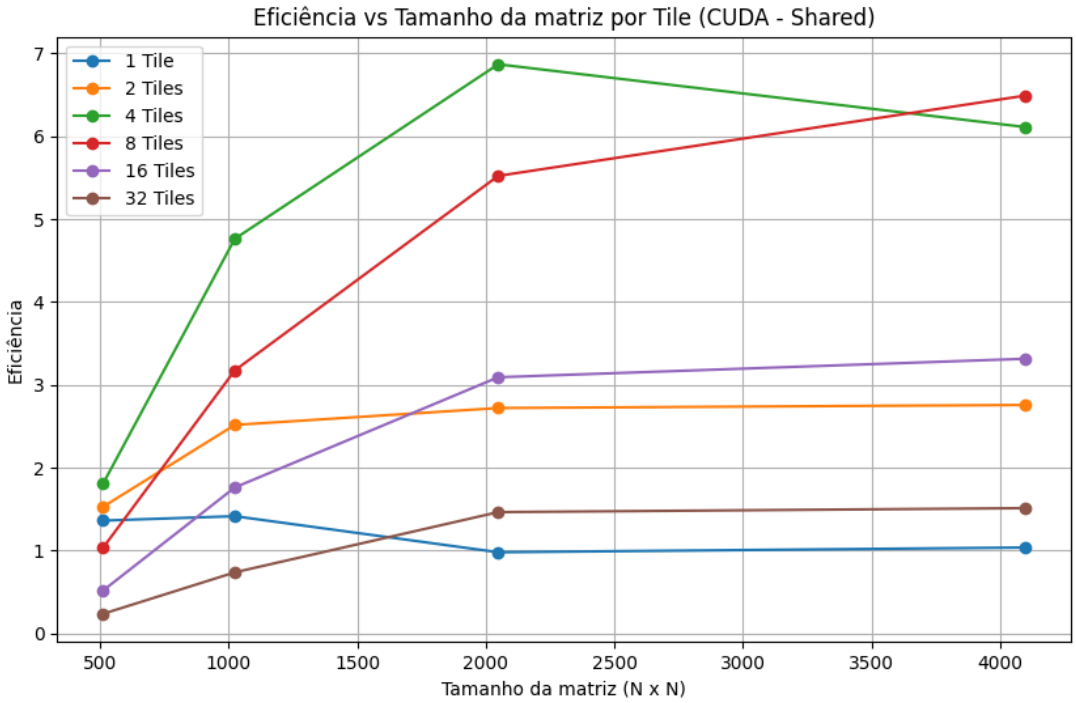

Figura 12: Eficiência (CUDA - Shared)

As Figuras 9 e 10 mostram a eficiência das implementações com OpenMP e MPI, respectivamente. A eficiência tende a diminuir com o aumento do número de threads/processos, o que é esperado devido à sobrecarga de comunicação e sincronização. Não há uma diferença significativa na eficiência entre as duas implementações, embora a implementação com OpenMP apresente uma leve vantagem em alguns casos. Já as Figuras 11 e 12 mostram a eficiência das implementações com CUDA (básica e otimizada com memória compartilhada). A eficiência da versão shared é pior em casos quando as matrizes tendem a crescer. Para o grafico da versao estendida de 32 tiles, o teste aplicado para a eficiencia

3.3 Testando os limites da GPU

A versao basica era superior a versão shared na maioria dos testes, então decidimos testar a versao shared com tiles de 32x32 (1024 threads por bloco) e a versão básica para tentar maximizar o uso da memoria e ver se haveria uma melhora significativa.

A figura abaixo mostra o gerenciador de tarefas do Windows, onde podemos ver o uso da GPU atingindo seu limite de quase 100% de uso durante a execução.

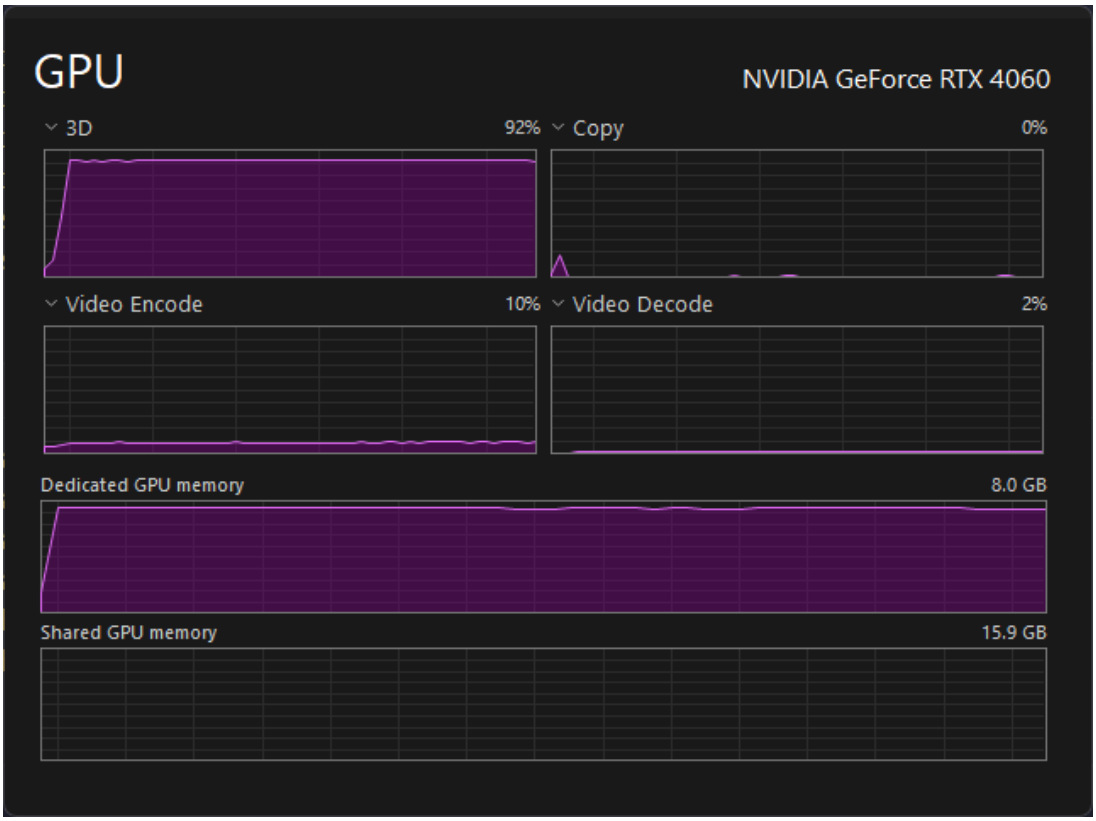

Figura 13: Eficiência (CUDA - 32 Tiles)

Ainda assim, o tempo de execução da versão shared com 32 tiles não superou a versão básica.

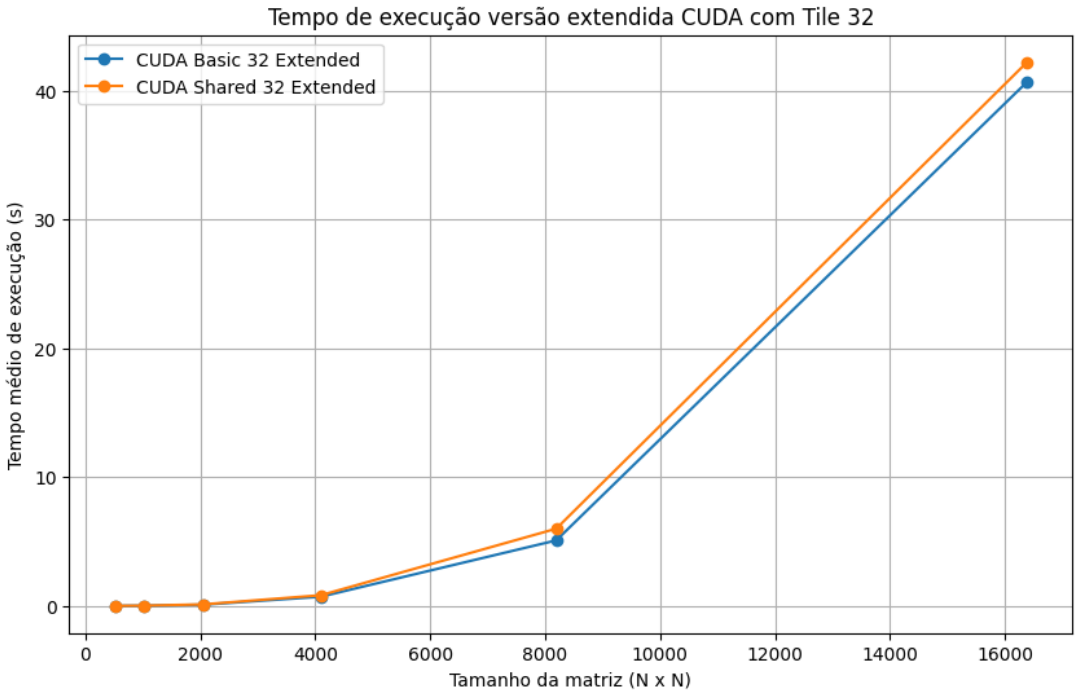

Figura 14: Tempo de execução (CUDA - 32 Tiles)

4 Discussão

Nesta seção, discutimos os resultados obtidos nas implementações sequencial, paralela com cuda da multiplicação de matrizes DGEMM. Analisamos o desempenho, a escalabilidade e as limitações de cada abordagem.

4.1 Análise de Desempenho

A implementação com CUDA Basic apresentou um desempenho superior em comparação com a implementação com MPI e OpenMP para Eficiência, speedup e tempo de execucao. O que foi contraditório à versão shared que teve um desempenho inferiror. O fato da GPU servir para esse proposito já era esperado que houvesse alguns ganhos nas execuções, mas o fato da versão shared não ter superado a básica foi inesperado.

4.2 Escalabilidade

A implementação com CUDA demonstrou uma boa escalabilidade com o aumento do tamanho das matrizes. O speedup aumentou de forma consistente com o aumento do número de threads, embora a eficiência tenha diminuído ligeiramente devido à sobrecarga de comunicação.

4.3 Limitações

Não houve limitações significativas observadas nas implementações. Mas o calculo da eficiencia é injusto com os outros métodos de processamento paralelo, pois a GPU possui milhares de núcleos, e o calculo da eficiencia é baseado no numero de threads (que é limitado a 1024 por bloco).

5 Conclusão

Neste relatório, apresentamos a implementação e análise de desempenho da multiplicação de matrizes DGEMM utilizando GPGPUs com CUDA, comparando os resultados com as abordagens de CPU (Sequencial, OpenMP e MPI) desenvolvidas nos projetos anteriores.

Os resultados demonstraram que a utilização de GPUs oferece um ganho de desempenho expressivo em relação às implementações baseadas em CPU, especialmente para matrizes de grande porte, validando a eficácia do paralelismo massivo para operações de álgebra linear densa. Um achado notável foi o comportamento da versão CUDA com memória compartilhada (*tiling*), que, contrariando a expectativa teórica inicial, obteve desempenho inferior à versão básica na GPU utilizada (RTX 4060). Atribuímos esse fenômeno à eficiência da hierarquia de cache moderna (L2 de grande capacidade) da arquitetura Ada Lovelace, que mitigou os gargalos de acesso à memória global automaticamente, tornando a sobrecarga de gerenciamento manual da memória compartilhada desvantajosa neste cenário específico.

Em trabalhos futuros, sugere-se a exploração de *CUDA Streams* para sobrepor a transferência de dados (PCIe) com a computação, bem como o uso de bibliotecas altamente otimizadas como cuBLAS ou a utilização de Tensor Cores para maximizar ainda mais o *throughput* da GPU.

Referências

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

[2] OpenMP, Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-RefGuide-6.0-OMP60SC24-web.pdf. Acesso em: 22 de Setembro de 2025.

[3] Open MPI, Disponível em: https://www.open-mpi.org/doc/v4.1/. Acesso em: 11 de Novembro de 2025.

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

[5] VSP-BERLIN, Disponível em: https://svn.vsp.tu-berlin.de/repos/public-svn/publications/kn-old/strc/html/node9.html. Acesso em: 23 de Setembro de 2025.

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

[7] NVIDIA CUDA C Programming Guide, Disponível em: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Acesso em: 24 de Novembro de 2025.