UNIVERSIDADE ESTADUAL DE SANTA CRUZ

BEATRIZ SANTOS DE OLIVEIRA E JÚLIA DE ARAÚJO RAMOS

Processamento paralelo: Projeto 3 — Multiplicação de Matrizes (DGEMM) com CUDA

ILHÉUS - BAHIA

2025

BEATRIZ SANTOS DE OLIVEIRA E JÚLIA DE ARAÚJO RAMOS

Processamento paralelo: Projeto 3 — Multiplicação de Matrizes (DGEMM) com CUDA

Trabalho avaliativo da disciplina Processamento paralelo ministrada pelo professor Esbel Tomás Valero Ollerana.

ILHÉUS - BAHIA
2025

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O presente trabalho constitui a terceira etapa de uma série de projetos focados na análise de desempenho e otimização de algoritmos paralelos, tendo como objeto de estudo central a multiplicação de matrizes de precisão dupla (DGEMM - Double-Precision General Matrix Multiplication). Nas etapas anteriores, foram exploradas as capacidades de processamento das Unidades Centrais de Processamento (CPUs) através de dois modelos distintos de execução:

- **Memória Compartilhada:** Utilizando a API OpenMP para paralelismo em nível de threads dentro de um único nó.
- **Memória Distribuída:** Utilizando o padrão MPI (*Message Passing Interface*) para comunicação entre processos em diferentes nós de processamento.

Dando continuidade a essa investigação, este projeto estende os conhecimentos adquiridos para o paradigma de programação em GPGPU (General Purpose GPU Programming) (NVIDIA Corporation 2025). Utilizando a arquitetura CUDA, busca-se explorar o paralelismo massivo disponível nas unidades de processamento gráfico para a execução do DGEMM.

1.0.1 Objetivos do Trabalho

O objetivo central deste projeto é estender o conhecimento adquirido anteriormente para o desenvolvimento de uma implementação eficiente do algoritmo DGEMM utilizando a plataforma CUDA da NVIDIA.

Para atingir este objetivo geral, foram definidos os seguintes objetivos específicos:

- **Implementação e Otimização:** Desenvolver o algoritmo de multiplicação de matrizes em GPU, aplicando conceitos avançados de hierarquia de memória. Isso inclui o gerenciamento explícito de dados entre a memória global, a memória compartilhada (para redução de latência), a memória constante e o uso eficiente de registradores.
- **Análise Comparativa:** Avaliar o desempenho da versão em CUDA em relação às implementações anteriores (Sequencial, OpenMP e MPI). Serão analisadas métricas de Speedup, Eficiência e Escalabilidade.
- **Validação Numérica:** Assegurar a corretude dos resultados através de testes rigorosos, considerando as potenciais divergências causadas por erros de arredondamento em operações de ponto flutuante (IEEE 754) em arquiteturas distintas.

Ao final deste estudo, espera-se fornecer uma visão crítica sobre as vantagens e limitações de cada modelo de execução — OpenMP, MPI e CUDA — identificando em quais cenários a aceleração via GPU oferece o melhor custo-benefício computacional.

2 Metodologia

Esta seção detalha o ambiente computacional, as estratégias algorítmicas implementadas para a CPU e a GPU, e os critérios utilizados para a validação dos resultados e análise de desempenho.

2.1 Descrição do Hardware Utilizado

Os experimentos foram realizados em um nó de processamento híbrido. O código foi compilado utilizando o nvcc (NVIDIA CUDA Compiler) com a flag de otimização -O3 e direcionado para a arquitetura sm_75 (Turing). As especificações do ambiente são:

- **CPU:** AMD Ryzen 7 4800H (Clock: 2.9 GHz, 8 núcleos e 16 threads).
- **Memória RAM:** 16 GB.
- **GPU:** NVIDIA GeForce GTX 1650 Ti.
  - Arquitetura: Turing (Compute Capability 7.5).
  - Memória Global: 4 GB.
- **Software:** Sistema Operacional Windows 10, CUDA Toolkit 11.5.

2.2 Estratégia de Implementação e Paralelização

Para a análise comparativa, foram desenvolvidas três versões do algoritmo de multiplicação de matrizes $C = A \times B + C$, todas manipulando matrizes quadradas de dimensão $N \times N$ armazenadas em ordem row-major. Essas versões estão descritas a seguir:

2.2.0.1 Versão Sequencial (Host)

Como base de referência (baseline), implementou-se uma versão na CPU utilizando a técnica de **Loop Unrolling** com fator 4. Essa otimização reduz o overhead de controle do laço interno e melhora o uso do pipeline de instruções do processador, oferecendo uma comparação mais robusta do que uma implementação sequencial ingênua.

2.2.0.2 Versão CUDA Naive (Device - Implementação Ingênua)

A implementação inicial, denominada *Naive*, transpõe diretamente a lógica dos laços aninhados da CPU para a grade de threads da GPU. Nesta abordagem, a responsabilidade computacional é distribuída da seguinte forma:

Mapeamento Thread-Elemento: Cada thread, identificada pelas coordenadas (tx, ty) na grade 2D, é responsável por calcular exclusivamente um elemento $C_{row,col}$ da matriz resultante.

Padrão de Acesso à Memória: Para computar esse único elemento, a thread executa um laço de $k = 0$ até $N - 1$. Em **cada iteração** desse laço, a thread realiza duas leituras na Memória Global (uma para $A[row][k]$ e outra para $B[k][col]$) e executa duas operações de ponto flutuante (multiplicação e soma).

Esta estratégia apresenta limitações de desempenho devido a dois fatores arquiteturais:

1. **Baixa Intensidade Aritmética:** A GPU é um processador projetado para alta vazão (*throughput*), ideal para algoritmos que realizam muitos cálculos por dado carregado. Na versão Naive, a razão entre operações de ponto flutuante (FLOPS) e bytes transferidos é muito baixa (aproximadamente 1 FLOP por 8 bytes lidos em precisão dupla). Isso torna o kernel **limitado pela memória** (*Memory Bound*), onde as unidades de processamento (CUDA Cores) passam a maior parte do tempo ociosas aguardando dados da DRAM.

2. **Redundância de Acessos (Desperdício de Banda):** Não há reaproveitamento de dados entre threads vizinhas.

   - Todas as threads de uma mesma linha da grade leem a mesma linha da matriz A.
   - Todas as threads de uma mesma coluna da grade leem a mesma coluna da matriz B.

   Sem o uso de memória compartilhada ou caches, a mesma informação é requisitada à Memória Global N vezes, saturando o barramento de memória e gerando uma latência de centenas de ciclos de clock para cada requisição.

2.2.0.3 Versão CUDA Tiled (Device - Otimizada)

A versão otimizada visa mitigar o principal gargalo da implementação ingênua: a alta latência de acesso à Memória Global. Para isso, utiliza-se a técnica de **Tiling** (blocagem), que decompõe as matrizes A e B em sub-blocos que cabem na memória rápida do chip (Memória Compartilhada).

A estratégia de implementação pode ser dividida em três pilares:

Configuração da Grade: O espaço de execução foi mapeado em blocos de dimensão 32×32 threads ($TILE\_SIZE = 32$). Cada bloco de threads é responsável por

calcular um sub-bloco quadrado de 32×32 elementos da matriz resultante C. O total de 1024 threads por bloco foi escolhido para maximizar a ocupação dos multiprocessadores da GPU.

Carregamento Cooperativo (Coalesced Access): Diferente da versão ingênua, as threads não leem dados aleatoriamente. Em cada iteração do laço principal, as threads do bloco trabalham em conjunto para carregar um "azulejo"(tile) da matriz A e um da matriz B da Memória Global para a **Memória Compartilhada** (`__shared__`). Como a memória compartilhada reside no próprio chip (on-chip), seu acesso é ordens de grandeza mais rápido que a DRAM.

Sincronização e Cálculo: O algoritmo avança em etapas.

1. As threads carregam os dados para a memória compartilhada.
2. Uma barreira `__syncthreads()` garante que todo o bloco terminou o carregamento.
3. As threads realizam os cálculos parciais usando apenas os dados rápidos da memória compartilhada.
4. Uma segunda barreira `__syncthreads()` assegura que todos os cálculos terminaram antes que o bloco carregue novos dados na próxima iteração, evitando condições de corrida.

2.3 Métricas e Validação

Os testes foram conduzidos com matrizes de tamanhos $N = \{512, 1024, 2048, 4096\}$.

- **Medição de Tempo:** O tempo de execução (T) foi medido utilizando a função clock() da biblioteca padrão C. Para as versões em GPU, foi inserida a chamada cudaDeviceSynchronize() antes da marcação final do tempo, garantindo que a medição contabilizasse o término real do processamento assíncrono do kernel. O tempo inclui a transferência de dados Host-Device e Device-Host apenas onde explicitamente necessário para a lógica do algoritmo.

- **Desempenho (GFLOPS):** A capacidade de processamento foi calculada em Giga Floating Point Operations Per Second, dada pela fórmula:

$$GFLOPS = \frac{2 \times N^3}{T_{exec} \times 10^9}$$

- **Validação Numérica (Corretude):** Devido às diferenças na ordem das operações de ponto flutuante entre CPU e GPU, implementou-se um teste de erro relativo. Considerou-se o resultado válido se a diferença relativa máxima entre a matriz

da GPU (C_{cuda}) e a referência da CPU (C_{seq}) fosse inferior a um limiar aceitável ($\epsilon \approx 10^{-12}$), calculado como:

$$\Delta_{rel} = \max \left(\frac{|C_{seq} - C_{cuda}|}{|C_{seq}| + \epsilon} \right)$$

3 Resultados

Esta seção apresenta os resultados obtidos nos testes de corretude e na análise de desempenho. Para cada configuração (tamanho de matriz e número de processos), os programas foram executados três vezes, e o script de análise em Python (`graficos.py`) utilizou a **média** de tempo para os cálculos.

3.1 Análise de Corretude

Os resultados da validação numérica indicam que, embora não haja uma identidade bit-a-bit (erro zero) entre as matrizes geradas pela CPU e pela GPU, a implementação é correta. Os valores de Delta observados nos testes variaram consistentemente na ordem de magnitude de 10^{-16} (oscilando entre 6.33×10^{-16} e 8.72×10^{-16} nas amostras coletadas).

Essa divergência ínfima é esperada em ambientes de computação heterogênea. Ela ocorre porque a aritmética de ponto flutuante (IEEE 754) não é associativa; ou seja, a alteração na ordem das somas realizada pelo paralelismo massivo da GPU, somada às diferenças de arquitetura de hardware (como o uso de instruções *Fused Multiply-Add*), acumula erros de arredondamento distintos da execução sequencial na CPU. Contudo, como o erro reside no limite da precisão da máquina para variáveis *double*, conclui-se que o algoritmo foi implementado com sucesso.

3.2 Análise de Desempenho

Os resultados das médias de tempo, Speedup e Eficiência foram conduzidos variando a dimensão das matrizes N em $\{512, 1024, 2048, 4096\}$ e são apresentados e analisados em detalhes nas subseções a seguir.

3.2.1 Análise do Tempo de Execução

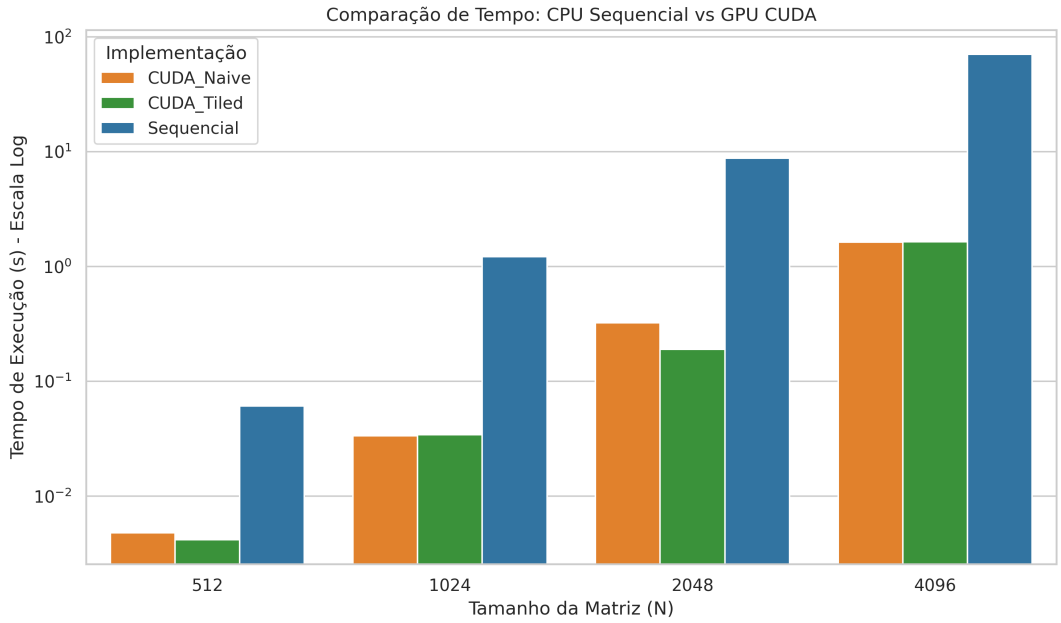

Figura 1 – Comparação do tempo de execução (em segundos) em escala logarítmica.

O gráfico apresentado na Figura 1 utiliza uma escala logarítmica no eixo vertical para permitir a visualização simultânea dos tempos da CPU (que crescem cúbica e abruptamente) e da GPU (que se mantêm na ordem de milissegundos ou poucos segundos).A análise dos resultados quantitativos revela três comportamentos distintos conforme a carga de trabalho aumenta:

Para dimensões menores, como $N = 512$ e 1024, embora a GPU apresente tempos inferiores aos da CPU, a diferença absoluta entre eles é pequena. Em $N = 512$, a execução sequencial levou aproximadamente 0.05 s, enquanto as versões GPU (*Naive* e *Tiled*) executaram em cerca de 0.004 s. Nesse cenário, a sobrecarga de lançamento do *kernel* e de transferência de dados torna-se proporcionalmente relevante. Além disso, não há diferença perceptível entre a versão *Naive* e a *Tiled*, pois os dados cabem confortavelmente nos caches L1 e L2 da GPU, tornando dispensável a otimização manual com uso de memória compartilhada.

No caso de matrizes de dimensão intermediária, como $N = 2048$, observa-se o impacto mais expressivo da técnica de *tiling*. A versão sequencial salta para 8.35 s, enquanto a versão CUDA *Naive* registra cerca de 0.38 s. A versão CUDA *Tiled* reduz esse tempo praticamente pela metade, alcançando 0.19 s. Esse ganho evidencia que, para esse tamanho de problema, o gargalo passa a ser o acesso direto à memória global da GPU. O *tiling* mitiga esse problema ao carregar blocos de dados para a memória compartilhada, de baixa latência, permitindo que as *threads* reutilizem dados sem recorrer repetidamente ao acesso lento à memória global.

Por fim, no maior caso de teste, $N = 4096$, a disparidade entre CPU e GPU torna-se extrema, reafirmando a importância do uso de aceleradores gráficos em aplicações de alto desempenho. A CPU necessitou de aproximadamente 67.8 s para realizar a multiplicação, enquanto as versões GPU concluíram a tarefa em cerca de 1.60 s (*Naive*) e 1.63 s (*Tiled*). Observa-se uma convergência de desempenho entre as duas versões em GPU, sugerindo que, nesse tamanho de matriz, o fator limitante deixa de ser predominantemente a latência de memória e passa a envolver outros recursos, como saturação da largura de banda da memória global, limitações térmicas da GPU ou esgotamento de registradores. Dessa forma, a otimização via *tiling* não traz ganhos significativos adicionais em relação ao *caching* automático da arquitetura Turing.

3.2.2 Análise do Speedup

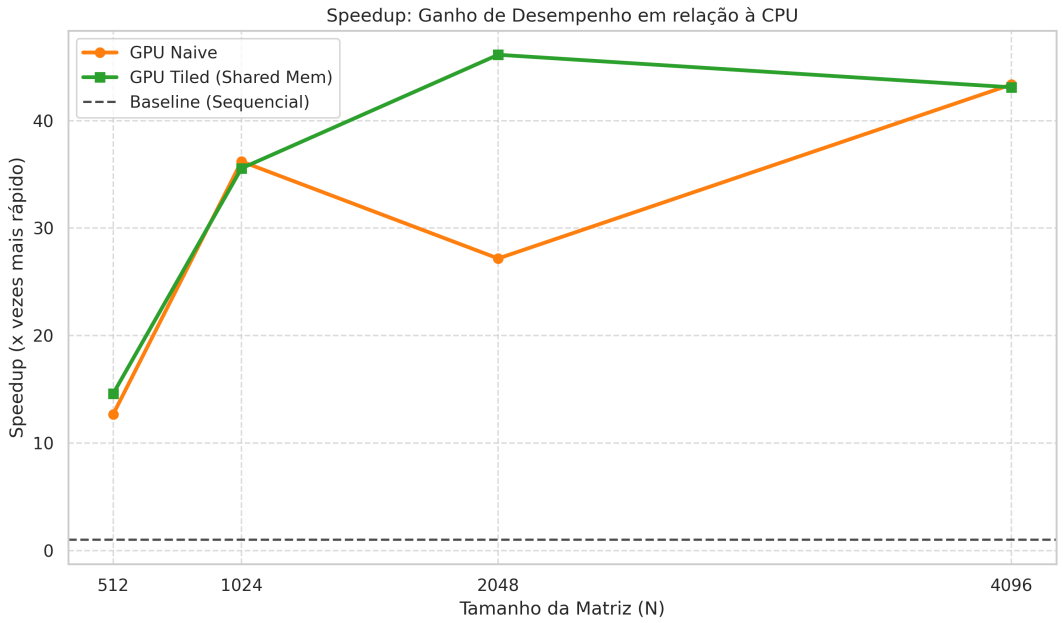

Figura 2 – Speedup: Ganho de desempenho em relação à CPU.

O Speedup (S) é a métrica que quantifica a aceleração relativa obtida pela paralelização, definida pela razão entre o tempo sequencial (T_{seq}) e o tempo paralelo (T_{cuda}):

$$S = \frac{T_{seq}}{T_{cuda}}$$

A Figura 2 ilustra as curvas de speedup para as implementações Naive (linha laranja) e Tiled (linha verde) em função do tamanho da matriz. A linha tracejada representa o baseline sequencial ($S = 1$). A análise do comportamento das curvas permite identificar três fases distintas de desempenho:

Para matrizes menores, na fase de crescimento entre $N = 512$ e 1024, observa-se um aumento acentuado na aceleração. Em $N = 512$, o speedup inicia em aproximadamente

12× a 14×. Embora significativo, este valor é limitado pela sobrecarga (overhead) de inicialização da GPU e transferências de memória PCIe, que possuem um peso proporcional maior quando o tempo de computação é muito curto. Ao dobrar o tamanho para $N = 1024$, o speedup salta para a casa de 36×, ponto em que a carga de trabalho torna-se suficiente para ocupar melhor os milhares de núcleos da GPU, diluindo o impacto do overhead.

O cenário de $N = 2048$ apresenta-se como o ponto crítico da análise, onde as curvas das duas implementações divergem significativamente, evidenciando o impacto da memória compartilhada e o gargalo de memória. Na versão CUDA Naive, a curva laranja sofre uma queda abrupta, reduzindo o speedup para aproximadamente 27×. Isso indica que, para este tamanho de matriz, o cache L2 da GPU não foi suficiente para conter os dados, forçando as threads a buscarem dados repetidamente na lenta Memória Global, tornando a aplicação limitada pela memória (Memory Bound). Em contraste, na versão CUDA Tiled, observa-se um pico de eficiência onde a curva verde atinge seu máximo, ultrapassando 45× de aceleração. A técnica de Tiling manteve os dados ativos na Memória Compartilhada (on-chip), evitando a latência da memória global que penalizou a versão Naive.

Para o maior tamanho testado ($N = 4096$), as duas curvas voltam a convergir para um patamar de aceleração em torno de 43×. Neste estágio de alta carga, a quantidade massiva de operações aritméticas e o volume total de dados possivelmente saturam a largura de banda máxima da placa ou os recursos de computação, fazendo com que a vantagem do gerenciamento manual de memória (Tiling) seja menos perceptível frente à eficiência dos escalonadores de hardware da arquitetura Turing. Em conclusão, os resultados demonstram que a GPU oferece um ganho de desempenho consistente acima de 40× para cargas de trabalho significativas. A implementação Tiled provou ser mais robusta, mantendo a estabilidade do desempenho mesmo em cenários ($N = 2048$) onde a implementação ingênua sofreu degradação devido à latência de memória.

3.2.3 Análise da Eficiência

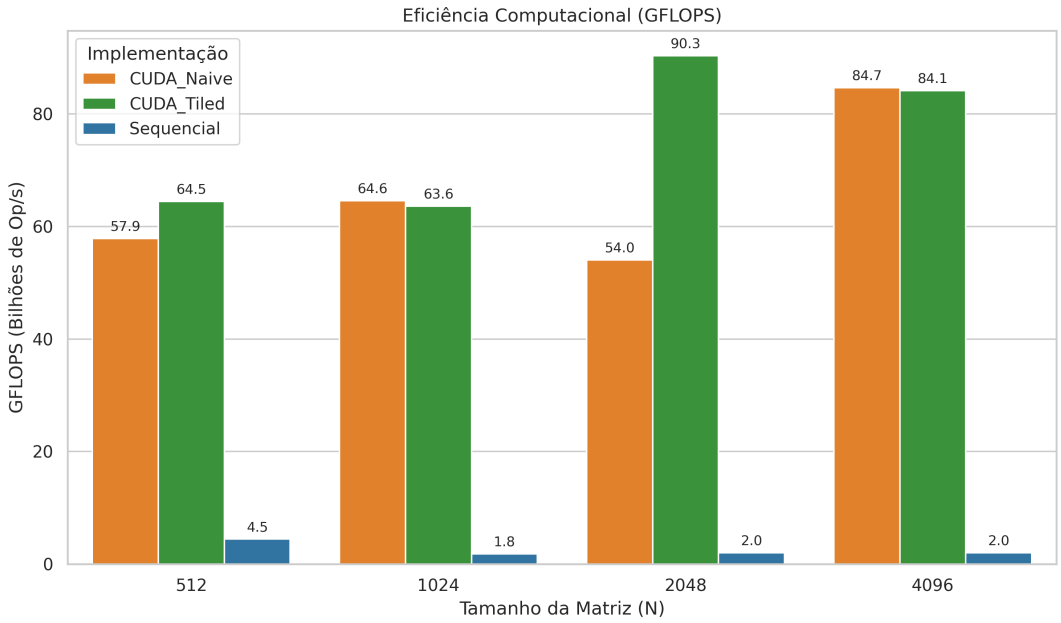

Figura 3 – Eficiência do código Cuda.

A análise da eficiência computacional, medida em bilhões de operações de ponto flutuante por segundo (GFLOPS), revela a capacidade de processamento sustentada pelas diferentes abordagens. Nas configurações de menor escala, compreendendo as matrizes de dimensão 512 e 1024, observa-se imediatamente a superioridade da arquitetura da GPU sobre a CPU. Enquanto a execução sequencial oscilou entre 1,8 e 4,5 GFLOPS, as implementações em CUDA saltaram para a faixa de 57 a 64 GFLOPS. Especificamente em $N = 512$, a versão Tiled apresentou uma leve vantagem (64,5 GFLOPS) sobre a Naive (57,9 GFLOPS), mas em $N = 1024$ o desempenho de ambas se equiparou em torno de 64 GFLOPS. Neste estágio, a carga de trabalho ainda não é suficiente para saturar completamente os recursos da placa, e os caches automáticos (L1/L2) da GPU conseguem gerenciar o acesso aos dados da versão ingênua com eficácia similar à otimização manual.

O comportamento em $N = 2048$ representa o ponto de inflexão da análise, onde a limitação da largura de banda da memória global se torna crítica e a técnica de Tiling demonstra seu verdadeiro valor. Neste cenário, a versão Naive sofreu uma queda de desempenho significativa, reduzindo sua vazão para 54,0 GFLOPS, o que indica que o algoritmo se tornou limitado pela memória (Memory Bound), com as unidades de processamento ociosas aguardando dados da DRAM. Em contrapartida, a versão Tiled atingiu seu pico de eficiência, saltando para impressionantes 90,3 GFLOPS. Esse ganho expressivo comprova que o uso explícito da memória compartilhada permitiu reutilizar os dados dentro do chip (on-chip), reduzindo drasticamente a latência e mantendo os núcleos de processamento alimentados constantemente.

Para o maior tamanho de matriz testado ($N = 4096$), observa-se uma convergência nos resultados das duas implementações em GPU. A versão Naive recuperou-se, atingindo 84,7 GFLOPS, enquanto a versão Tiled estabilizou-se em 84,1 GFLOPS. Neste patamar de carga massiva, a GPU opera próxima de seus limites de saturação, seja por largura de banda total ou por restrições térmicas e de escalonamento de warps. O fato de ambas as versões entregarem cerca de 84 GFLOPS — um valor mais de 40 vezes superior aos 2,0 GFLOPS mantidos pela CPU sequencial — reafirma que, para problemas de larga escala, o paralelismo massivo da GPU é a solução dominante, embora a vantagem relativa da otimização manual de memória (Tiling) possa diminuir quando o hardware atinge seu limite físico de vazão global.

4 Discussão

4.1 Comparativo de Tempos de Execução (OpenMP vs. MPI vs. CUDA)

A Tabela 1 consolida os melhores tempos de execução obtidos para a multiplicação de matrizes para cada tamanho de matriz, confrontando os três paradigmas de programação paralela explorados ao longo dos projetos.

Tabela 1 – Comparativo de Tempos na Melhor Configuração por Paradigma

Matriz (N)	**Paradigma**	**Melhor Configuração**	**Tempo (s)**
512	MPI	8 Processos	0.01
	OpenMP	12 Threads	0.00
	CUDA_Naive		0.0048
	CUDA_Tiled		0.0041
1024	MPI	4 Processos	0.17
	OpenMP	12 Threads	0.02
	CUDA_Naive		0.033
	CUDA_Tiled		0.034
2048	MPI	8 Processos	2.14
	OpenMP	10 Threads	0.29
	CUDA_Naive		0.32
	CUDA_Tiled		0.19
4096	MPI	8 Processos	17.98
	OpenMP	10 Threads	2.73
	CUDA_Naive		1.62
	CUDA_Tiled		1.63

Nos cenários de pequena escala, compreendendo matrizes de dimensões 512 e 1024, observa-se que o desempenho é limitado predominantemente pela latência e pelo overhead de gerenciamento, e não pelo poder de processamento bruto. O MPI apresentou o desempenho mais modesto nessas configurações, uma vez que o custo temporal para iniciar os processos e trocar mensagens com fatias pequenas da matriz superou o tempo gasto no cálculo propriamente dito. Em contrapartida, as implementações em OpenMP e CUDA apresentaram um empate técnico, com tempos na ordem de milissegundos. Embora a GPU possua maior capacidade teórica, para problemas deste porte, a vantagem é mascarada pelo custo de transferência de dados via barramento PCI-Express e pela inicialização do kernel, que consomem uma fração de tempo comparável à execução do algoritmo em si.

O cenário muda drasticamente à medida que a dimensão do problema aumenta para escalas maiores ($N = 2048$ e 4096), onde a complexidade cúbica do algoritmo ($O(N^3)$)

faz com que a capacidade de vazão (throughput) se torne o fator determinante. Para o maior caso de teste ($N = 4096$), a GPU consolidou-se como a arquitetura dominante, atingindo a marca de 1,62 segundos na versão Naive e 1,63 segundos na versão Tiled. A natureza massivamente paralela do dispositivo, com milhares de threads ativas ocultando a latência de acesso à memória, permitiu processar o grande volume de operações de ponto flutuante com uma eficiência inalcançável pelas CPUs tradicionais.

Ao confrontar os melhores tempos absolutos para a matriz de dimensão 4096, estabelece-se uma hierarquia clara de desempenho. A solução em OpenMP se mostrou eficiente, finalizando a tarefa em 2,73 segundos; este resultado indica que o processador multicore conseguiu utilizar eficazmente seus caches para minimizar o impacto da menor largura de banda da memória RAM. Já o MPI, mesmo em sua melhor configuração, registrou 17,98 segundos, sendo aproximadamente 11 vezes mais lento que a GPU e 6,5 vezes mais lento que o OpenMP. Essa disparidade evidencia que, para álgebra linear densa executada em um único nó, a penalidade de comunicação do modelo distribuído cria um gargalo insuperável frente às arquiteturas de memória compartilhada e aceleradores gráficos. Em suma, para a classe de problemas testada, a GPU oferece o melhor retorno de desempenho, seguida de perto pelo OpenMP, restando ao MPI a utilidade em cenários onde a memória de um único dispositivo não seria suficiente.

4.2 Comparativo de Speedup

A análise das curvas de escalabilidade apresentadas nas figuras referentes ao CUDA, OpenMP e MPI revela comportamentos distintos entre os paradigmas estudados.

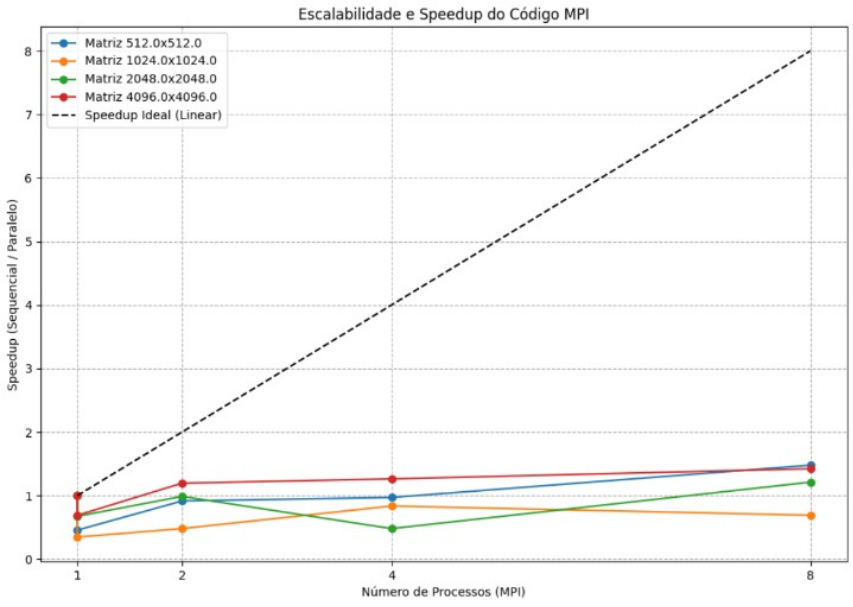

Figura 4 – Escalabilidade e Speedup do código MPI.

No caso do MPI (Figura 4), o gráfico de escalabilidade demonstra a dificuldade inerente de paralelizar algoritmos densos em arquiteturas de memória distribuída sem o suporte de uma rede de interconexão de alta performance. Observa-se uma curva sublinear onde, para a matriz de 4096×4096, o speedup com 8 processos foi de apenas $\approx 1.5\times$, mantendo-se muito distante da reta ideal. O gargalo de comunicação mostrou-se crítico, uma vez que o custo de transmitir grandes blocos de dados entre processos anulou quase todo o ganho obtido pela divisão do processamento. Em cenários com menos processos (1 e 2), o desempenho foi inclusive inferior ou igual ao sequencial ($S \leq 1$), indicando que o overhead do ambiente MPI pesou mais que o cálculo em si.

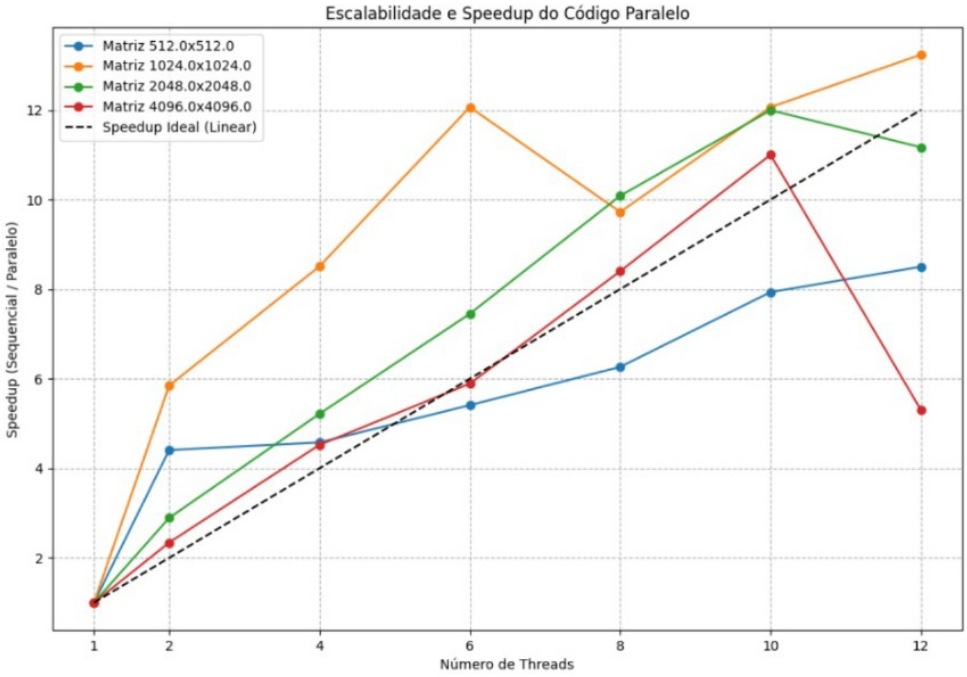

Figura 5 – Escalabilidade e Speedup do código OpenMP.

Em contraste, o comportamento do OpenMP (Figura 5) mostrou-se extremamente eficiente para a arquitetura multicore. Observa-se que a curva da matriz 4096×4096 ultrapassa a linha de "Speedup Ideal"em vários pontos; com 10 threads, por exemplo, obteve-se um $S \approx 11\times$. Esse fenômeno, conhecido como speedup superlinear, ocorre porque, ao utilizar 10 núcleos, a quantidade total de memória cache (L1/L2) disponível para o problema aumenta, reduzindo a necessidade de acesso à memória RAM lenta em comparação à execução sequencial em um único núcleo. Contudo, nota-se um ponto de saturação com uma queda brusca de desempenho ao passar de 10 para 12 threads, o que sugere que o hardware atingiu seu limite físico (provavelmente o número de núcleos físicos da máquina), e o uso de Hyper-Threading ou a disputa por recursos de memória começou a degradar a performance.

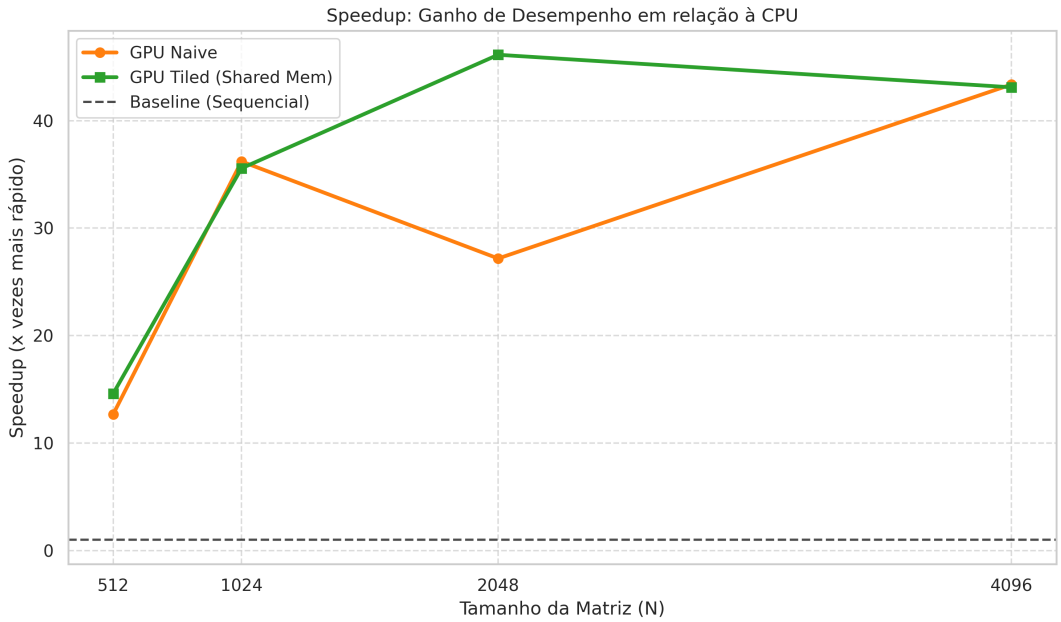

Figura 6 – Escalabilidade e Speedup do código Cuda.

Já o gráfico de speedup da GPU (Figura 6) apresenta uma ordem de magnitude diferente. Enquanto o MPI e o OpenMP oscilam entre ganhos de 1.5× a 11×, a GPU entrega consistentemente acelerações acima de 40× para grandes matrizes. Diferente do OpenMP, que degradou com o aumento excessivo de threads, a GPU manteve o desempenho alto em $N = 4096$ ($\approx 43\times$), comprovando que a arquitetura massivamente paralela é ideal para problemas de álgebra linear densa (Compute Bound). A versão Tiled se mostrou essencial para manter essa escalabilidade em tamanhos intermediários ($N = 2048$), cenário em que a versão Naive sofreu com gargalos de memória.

4.3 Comparativo de Eficiência

A análise comparativa da eficiência busca compreender o quão bem cada paradigma utiliza os recursos computacionais disponíveis. É importante ressaltar, preliminarmente, que esta seção confronta exclusivamente os resultados obtidos com MPI (Memória Distribuída) e CUDA (GPU), uma vez que os dados referentes à curva de eficiência do OpenMP não foram gerados no escopo do Projeto 1.

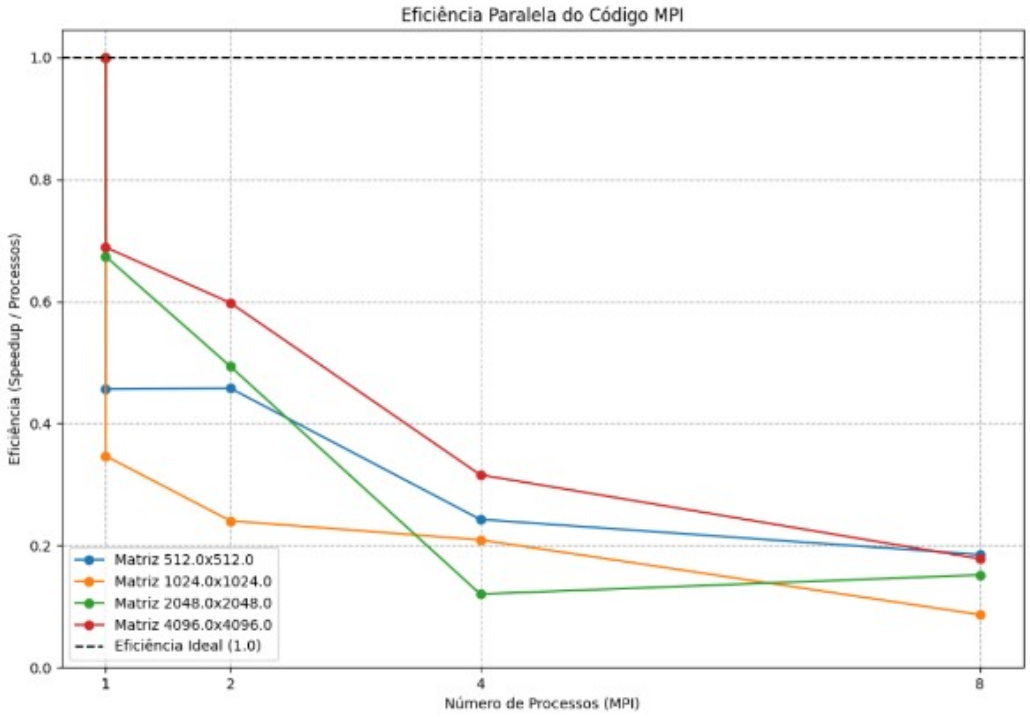

Figura 7 – Eficiência do código MPI.

Ao examinar o gráfico de eficiência do MPI (Figura 7), observa-se um comportamento de decaimento acentuado característico de sistemas distribuídos limitados por comunicação. A eficiência, definida neste contexto como a razão entre o speedup e o número de processos ($E = S/P$), degrada-se rapidamente à medida que mais recursos são adicionados. Para a maior matriz testada ($N = 4096$), a eficiência com 8 processos caiu para valores inferiores a 0,2 (ou 20%). Isso indica que 80% do poder computacional adicionado foi desperdiçado em overhead de sincronização e troca de mensagens, evidenciando que, para a granularidade deste problema, o custo de comunicação via rede penaliza severamente o aproveitamento dos núcleos extras da CPU.

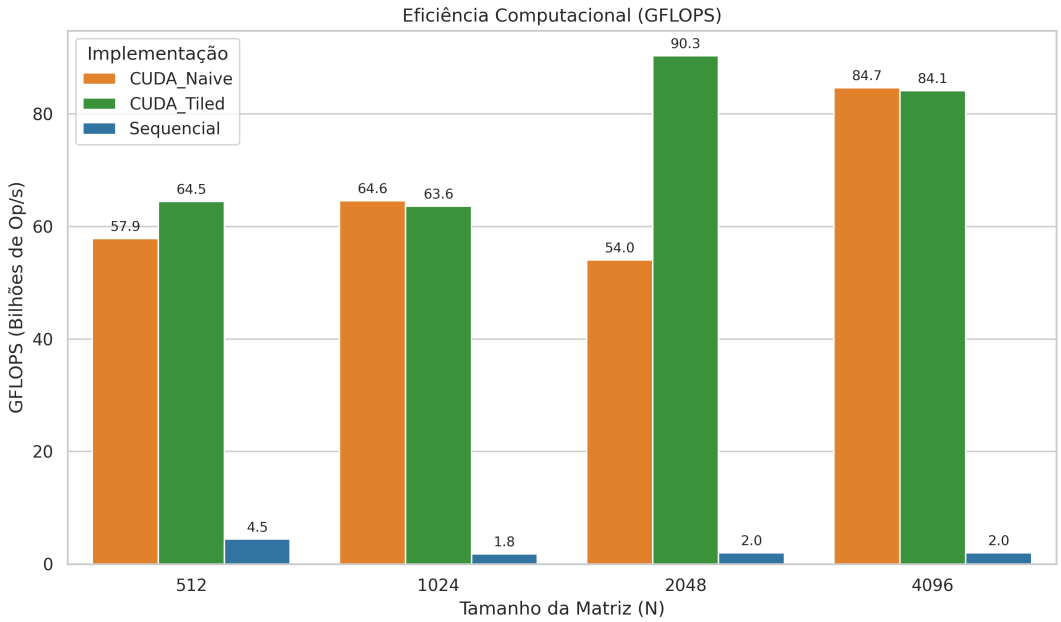

Figura 8 – Eficiência do código Cuda.

Em contraste oposto, a análise da eficiência da GPU (apresentada em termos de vazão computacional ou GFLOPS) (Figura 8) revela uma curva de crescimento logarítmico que se estabiliza em patamares elevados. Enquanto o MPI perde eficiência ao tentar escalar, a arquitetura CUDA ganha eficiência conforme a carga de trabalho aumenta. Para matrizes pequenas ($N = 512$), a GPU é subutilizada; contudo, ao atingir escalas maiores ($N = 4096$), a eficiência salta para cerca de 84 GFLOPS, demonstrando que o dispositivo requer um volume massivo de dados para saturar seus milhares de núcleos e justificar a latência de transferência.

Conclui-se, portanto, que os dois paradigmas enfrentam desafios opostos de escalabilidade. O MPI sofre de "escalabilidade forte"pobre para este algoritmo em um único nó, onde dividir o trabalho gera mais custo do que benefício. Já a GPU demonstra excelente "escalabilidade fraca"implícita: quanto maior o problema, melhor ela consegue esconder as latências de memória e maximizar o uso do hardware, entregando uma densidade de processamento que o modelo de troca de mensagens não conseguiu acompanhar neste experimento.

5 Conclusão

O desenvolvimento deste terceiro projeto permitiu consolidar a compreensão prática sobre a Computação de Alto Desempenho em aceleradores gráficos. Ao implementar o algoritmo de multiplicação de matrizes (DGEMM) utilizando a plataforma CUDA, foi possível observar não apenas o potencial de processamento bruto das GPUs, mas, principalmente, a importância crítica do gerenciamento da hierarquia de memória.

Os objetivos propostos foram plenamente alcançados. A implementação em GPU demonstrou-se funcional e numericamente correta, apresentando divergências infinitesimais ($\Delta \approx 10^{-16}$) esperadas devido à aritmética de ponto flutuante em arquiteturas paralelas. Em termos de desempenho, a GPU provou ser a arquitetura mais eficiente para problemas de álgebra linear densa, atingindo um *speedup* superior a $40\times$ em relação à execução sequencial e superando largamente as implementações anteriores com OpenMP e MPI.

A análise comparativa evidenciou como a hierarquia de memória impacta o desempenho final. A implementação otimizada (*Tiled*), que faz uso explícito da memória compartilhada (`__shared__`), mostrou-se essencial para sustentar a eficiência em tamanhos intermediários ($N = 2048$), contornando a latência da memória global que limitou a versão ingênua (*Naive*). A organização das threads em blocos de 32×32 permitiu maximizar a ocupação dos multiprocessadores, garantindo que a GPU fosse utilizada de forma efetiva como um dispositivo de computação massiva, e não apenas como um repositório de dados.

Por fim, no que tange à relação entre comunicação e computação, o modelo adotado utilizou transferências síncronas entre *Host* e *Device*. Embora isso tenha gerado um impacto perceptível em matrizes pequenas ($N = 512$), onde o tempo de cópia foi proporcionalmente relevante, para grandes volumes de dados ($N = 4096$) a intensidade aritmética do algoritmo foi suficiente para diluir esse custo. Conclui-se que o domínio das técnicas de *tiling* e o entendimento da arquitetura de memória são os fatores determinantes para extrair o desempenho máximo das modernas unidades de processamento gráfico.

Referências

NVIDIA Corporation. *CUDA Toolkit Documentation*. [S.l.], 2025. Version 13.0 Update 2. Disponível em: <https://docs.nvidia.com/cuda/>. Citado na página 9.