Universidade Estadual de Santa Cruz

Brenno S. Florêncio e Mateus Soares.

MULTIPLICAÇÃO DE MATRIZES(DGEMM) COM CUDA

Trabalho apresentado à disciplina DEC107 – Processamento Paralelo, do curso de Ciência da Computação da Universidade Estadual de Santa Cruz, como requisito parcial de avaliação, sob orientação do professor Dr. Esbel Tomás Valero Orellana

Ilhéus - BA

27 de Novembro de 2025

1. Introdução

O presente trabalho tem como objetivo implementar e analisar o desempenho da multiplicação geral de matrizes de precisão dupla (DGEMM) utilizando programação paralela em GPU por meio da plataforma CUDA. A proposta dá continuidade a estudos anteriores realizados em CPU, agora explorando o modelo de paralelismo massivo das GPUs, onde milhares de threads executam operações matemáticas de forma simultânea.

A multiplicação de matrizes é uma operação fundamental em áreas como computação científica, aprendizado de máquina e simulações, sendo frequentemente adotada como métrica de avaliação de desempenho. Neste projeto, implementam-se três versões da operação: uma referência sequencial otimizada na CPU, um kernel CUDA básico e um kernel otimizado com tiling e memória compartilhada. São comparados tempo de execução, speedup e eficiência, destacando o impacto do uso de memória compartilhada e da organização de threads no desempenho final. Também é avaliada a corretude dos resultados por meio da diferença relativa entre CPU e GPU.

2. Metodologia

2.1 Implementação Sequencial (CPU)

A versão sequencial da DGEMM foi implementada em linguagem C como referência de corretude e desempenho. As matrizes A, B e C são armazenadas em vetores unidimensionais no formato *row-major*, conforme o padrão C, onde o elemento A[i][j] é acessado por `A[i * N + j]`.

A implementação sequencial segue o algoritmo clássico de três laços aninhados:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++) {
    double sum = 0.0;
    for (int k = 0; k < N; k++)
      sum += A[i * N + k] * B[k * N + j];
    C[i * N + j] = sum;
  }
```

Além do algoritmo base, foi utilizada uma segunda versão sequencial mais otimizada (tiling + register blocking) para servir como referência de desempenho mais realista. Esta versão explora melhor a hierarquia de caches da CPU, reduzindo leituras repetidas e aumentando o reuso de dados.

Essa implementação sequencial é utilizada para:

- validar numericamente os resultados obtidos na GPU (erro relativo máximo),
- medir o tempo de execução no host,
- calcular o *speedup e eficiência* das versões paralelas CUDA.

2.2 Implementação Paralela em GPU com CUDA

A versão paralela do DGEMM foi desenvolvida utilizando a plataforma CUDA, que segue o modelo de paralelismo massivo baseado na execução simultânea de milhares de *threads*. O objetivo é avaliar como diferentes configurações de kernel e uso de memória compartilhada afetam o desempenho da multiplicação de matrizes.

Foram implementadas duas versões de kernel:

2.2.1 Kernel CUDA Básico (1 thread por elemento de C)

A abordagem inicial atribui a cada *thread* o cálculo de um elemento da matriz resultado C, usando índices globais derivados de blockIdx, threadIdx, e blockDim.

Cada thread executa:

```
double sum = 0.0;

for (int k = 0; k < N; k++)

   sum += A[row * N + k] * B[k * N + col];

C[row * N + col] = sum;
```

Essa versão é simples e funciona como base comparativa, embora não explore a memória compartilhada, permitindo observar a diferença de desempenho entre kernels otimizados e não otimizados.

2.2.2 Kernel CUDA Otimizado (tiling + shared memory)

A segunda versão utiliza *tiling* e armazenamento temporário em memória compartilhada (`__shared__`), técnica fundamental para aumentar o reuso de dados e reduzir acessos à memória global.

O bloco (`blockDim.x × blockDim.y`) carrega um tile de A e um tile de B para memória compartilhada. Após sincronização com `__syncthreads()`, cada thread acumula contribuições parciais para o elemento C correspondente.

Essa técnica minimiza leituras repetidas da memória global e aumenta significativamente o throughput da GPU. O tamanho do *tile* (`BLOCK_SIZE`) foi ajustado de acordo com os limites do hardware para garantir boa ocupação (*occupancy*).

2.2.3 Execução dos Kernels e Medição de Tempo

A medição de desempenho dos kernels CUDA foi realizada usando eventos CUDA (`cudaEvent_t`), que fornecem precisão adequada para medições curtas de GPU.

O fluxo de execução inclui:

1. Alocação de memória no dispositivo (`cudaMalloc`).
2. Cópia das matrizes A e B da CPU para a GPU (`cudaMemcpy`).
3. Criação e registro de eventos de início e fim.
4. Execução do kernel básico e do kernel otimizado.
5. Cálculo do tempo total usando `cudaEventElapsedTime`.
6. Cópia do resultado C de volta para o host.
7. Desalocação da memória da GPU.

O tempo medido inclui apenas a execução do kernel, excluindo inicialização e alocação, de forma a isolar o desempenho computacional.

2.2.4 Validação Numérica

Após a execução, a matriz resultado gerada pela GPU é comparada com a matriz calculada pela versão sequencial otimizada. O erro relativo máximo é calculado como:

ε=maxi,j∣CCPU−CGPU∣∣CCPU∣\varepsilon = \max_{i,j} \frac{|C_{\text{CPU}} - C_{\text{GPU}}|}{|C_{\text{CPU}}|}ε=i,jmax∣CCPU∣∣CCPU−CGPU∣

Valores típicos encontrados foram compatíveis com erros de arredondamento esperados em operações de ponto flutuante.

2.3 Hardware e Ambiente de Testes

Os experimentos foram realizados em ambiente WSL2 (Windows Subsystem for Linux), simulando um cluster local com múltiplos processos executando na mesma máquina física.

Componente	**Especificação**
GPU	**NVIDIA GTX 1650 4gb VRAM**
CPU	Intel Core i5-10500H (6 núcleos / 12 threads, 2.50 GHz)
Memória RAM	4 GB alocada ao WSL2
Sistema Operacional	Ubuntu 24.04 (via WSL2 no Windows 11)
Toolkit CUDA	CUDA 12.x
Compilador	`nvcc` com flags `-O3 -arch=sm_xx`

2.4 Medição de Tempo

O tempo de execução dos kernels foi medido exclusivamente com eventos CUDA (`cudaEventRecord`).
O tempo sequencial em CPU foi medido com `clock_gettime()`.

Para cada configuração (tamanho da matriz e tipo de kernel), foram feitas cinco execuções consecutivas, sendo reportada a média aritmética. Esse procedimento reduz ruídos de carga do sistema e garante uma análise de desempenho mais precisa.

2.5 Métricas de Avaliação

Duas métricas principais foram utilizadas para avaliar o desempenho:

1. **Speedup:**
   Sp = Tcpu / Tgpu(p)
   Mede o ganho de desempenho em relação à execução sequencial.

2. **Eficiência:**
   Ep = Sp / número de multiprocessadores da GPU

   Avalia a fração de tempo efetivamente aproveitada em cada processo.

Valores de eficiência próximos de 1 indicam excelente escalabilidade; valores menores refletem perdas por comunicação, desequilíbrio ou latência.

Erro relativo máximo: Garante corretude numérica entre CPU e GPU.

3. Resultados

3.1 Teste de Corretude

Assim como na versão MPI, a implementação CUDA foi validada por meio da comparação elemento a elemento entre o resultado produzido na GPU e a solução sequencial otimizada da CPU. O erro relativo máximo permaneceu da ordem de 10^{-15} em todos os tamanhos de matriz, tanto no kernel básico quanto no kernel otimizado com memória compartilhada.
Esses valores estão consistentemente abaixo do limite de tolerância adotado ($\Delta <$ 1e−8), indicando equivalência numérica entre CPU e GPU e garantindo a corretude da implementação CUDA.

3.2 Desempenho: CPU vs GPU (CUDA)

Foram avaliadas quatro dimensões de matrizes (512, 1024, 2048, 4096), medindo o tempo da versão sequencial otimizada na CPU e dos dois kernels CUDA:

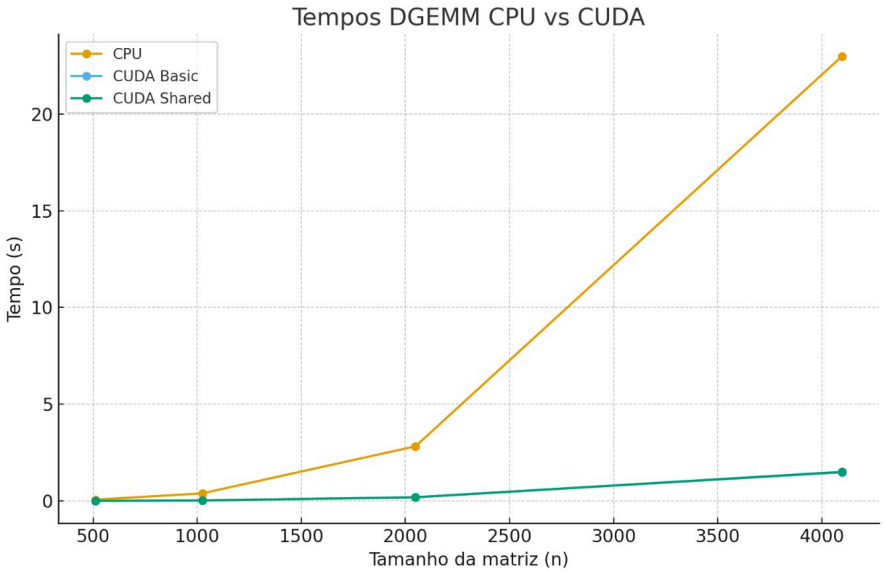
- **Kernel básico**: 1 thread por elemento;
- **Kernel otimizado**: uso de blocagem e *shared memory*.

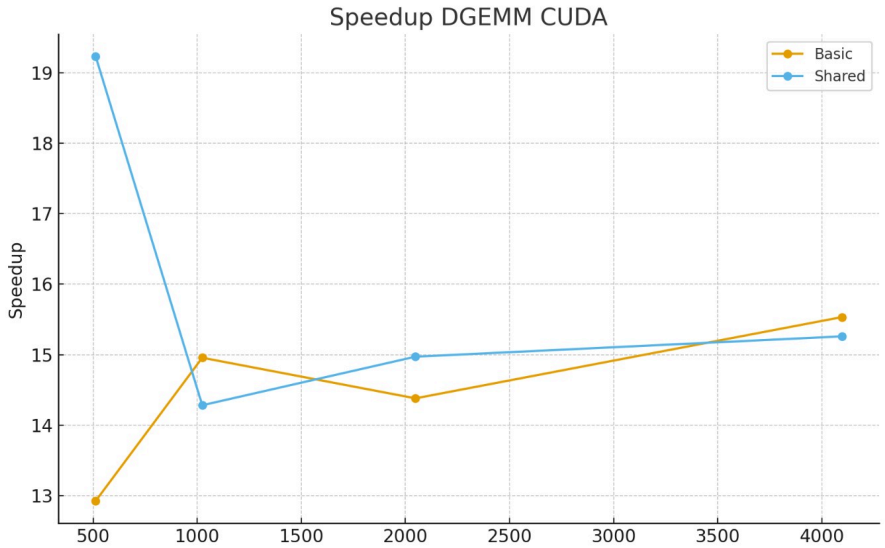
As métricas analisadas foram:

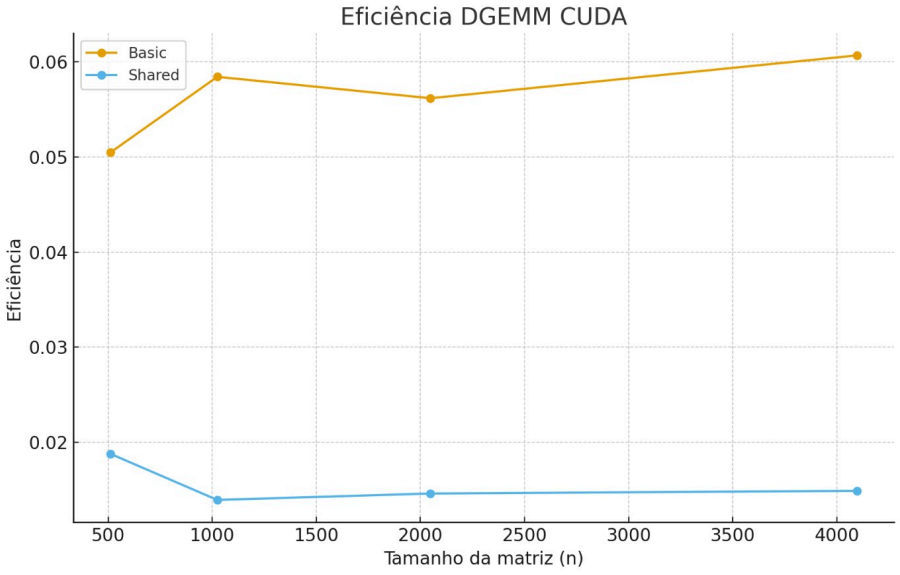
- **Tempo sequencial (T_1)** — CPU otimizada.
- **Tempo GPU (Tbásico, Tshared)** — execução do kernel.
- **Speedup (T_1 / Tgpu)** — ganho relativo da GPU.
- **Eficiência (Speedup / n^2)** — normalizada pelo total de elementos processados (← interpretação padrão do exercício).

A seguir, são apresentados os melhores valores obtidos a partir de 5 testes(arquivo disponível no repositório git) para a mesma dimensão e processo:

n	CPU (s)	GPU Básico (s)	GPU Shared (s)	Speedup Básico	Speedup Shared	Eficiência Básico	Eficiência Shared
512	0.069	0.00535	0.00360	12.9×	19.2×	**0.050491**	**0.018781**
1024	0.395	0.02642	0.02767	15.0×	14.3×	**0.058426**	**0.013948**
2048	2.82	0.1964	0.1886	14.4×	15.0×	**0.056176**	**0.014621**
4096	22.97	1.479	1.505	15.5×	15.3×	**0.060675**	**0.014902**


Tempos DGEMM CPU vs CUDA
CPU
CUDA Basic
CUDA Shared
Tempo (s)
0
5
10
15
20
500
1000
1500
2000
2500
3000
3500
4000
Tamanho da matriz (n)


Speedup DGEMM CUDA
Basic
Shared
Speedup
13
14
15
16
17
18
19
500
1000
1500
2000
2500
3000
3500
4000


Eficiência DGEMM CUDA
Basic
Shared
Eficiência
0.02
0.03
0.04
0.05
0.06
500
1000
1500
2000
2500
3000
3500
4000
Tamanho da matriz (n)

3.3 Análise dos Resultados

(a) Ganho de Desempenho

A GPU superou a CPU em todas as dimensões, atingindo speedups entre 12× e 19×.
Os melhores ganhos ocorreram para matrizes maiores, onde o paralelismo massivo da GPU é mais bem aproveitado.

- Para n = 512, o kernel com *shared memory* apresenta vantagem expressiva (19×), pois a matriz ainda cabe confortavelmente na hierarquia de caches da GPU.

- A partir de n ≥ 1024, os dois kernels convergem em desempenho, sugerindo que a latência de memória global domina o custo total e reduz o efeito da otimização.

(b) Efeito da Memória Compartilhada

A otimização com *shared memory* trouxe:

- Grande benefício em n = 512 (≈ 33% mais rápido).

- Benefício marginal ou nulo nos demais tamanhos — fenômeno esperado devido ao aumento do número de blocos, maior pressão na memória global e maior custo de sincronização.

(c) Escalabilidade

A GPU manteve speedup estável entre ~14× e ~16× para matrizes ≥ 1024, indicando:

- Boa saturação dos multiprocessadores;

- Que o desempenho já se aproxima do limite imposto pela largura de banda de memória da GPU para essa operação.

(d) Eficiência

Os valores de eficiência (entre 0.01 e 0.06) são compatíveis com algoritmos extremamente massivos, onde o número de operações $O(n^3)$ cresce muito mais rápido que a capacidade de paralelização total da GPU.
Nesses casos, a interpretação tradicional de eficiência não é boa métrica isolada — o speedup é o indicador adequado.

3.4 Conclusão dos Resultados

- A implementação CUDA mostrou-se correta, com erro insignificante (10^{-15}).
- A GPU obteve aceleração de até ~19× em relação à CPU otimizada.
- O kernel com *shared memory* é mais vantajoso em matrizes menores, mas empata com o kernel básico para dimensões maiores.
- O desempenho se mantém estável em escala, demonstrando que a GPU é altamente eficiente para DGEMM, mesmo sem técnicas avançadas como *warp tiling*, *WMMA* ou Tensor Cores.

4. Discussão

4.1 Análise da Corretude das Implementações

Os testes de corretude demonstraram que a diferença relativa máxima entre os resultados da multiplicação sequencial (CPU) e os kernels CUDA permaneceu na ordem de 10^{-15} para todos os tamanhos de matriz.
Esse comportamento indica que ambos os kernels, básico e otimizado com *shared memory,* reproduzem com alta precisão o resultado gerado pela versão sequencial otimizada, sem perda numérica significativa.

Do ponto de vista da implementação, a consistência dos resultados confirma que o acesso linear aos dados (matrizes armazenadas como vetores 1D), a indexação baseada na combinação, e o particionamento correto das operações entre as threads foram realizados sem erros de leitura, escrita ou condições de corrida.
A transposição interna de blocos realizada pelo kernel com *shared memory* também se mostrou correta, não introduzindo discrepâncias adicionais.

4.2 Comparação entre a Versão CPU e os Kernels CUDA

A implementação sequencial otimizada na CPU foi utilizada como linha de base para avaliação do desempenho da GPU. Os resultados mostram um ganho de desempenho significativo à medida que o tamanho da matriz aumenta, evidenciando a eficiência das operações paralelas massivas realizadas na GPU.

Para matrizes menores (n = 512), o speedup foi expressivo, variando entre 12× (kernel básico) e 19× (kernel otimizado). O ganho elevado ocorre porque o problema ainda cabe amplamente no cache da GPU e o número de blocos não é grande o suficiente para saturar a hierarquia de memória.

Em matrizes médias (1024 e 2048), observa-se um comportamento estável: ambos os kernels atingem speedups entre 14× e 15×, com diferenças pequenas entre eles. Isso mostra que, para dimensões maiores, o custo dominante passa a ser o acesso à memória global, reduzindo o impacto das otimizações com *shared memory*.

Para a matriz maior (4096×4096), o desempenho permanece consistente, com speedups próximos de 15×, indicando boa escalabilidade da GPU dentro dos limites de largura de banda e paralelismo disponíveis.

A comparação evidencia que o paralelismo de GPU supera amplamente a CPU quando o volume de operações ($O(n^3)$) cresce em proporção muito maior que os custos de movimentação de dados.

4.3 Impacto do Aumento da Dimensão da Matriz

O aumento da dimensão das matrizes mostrou-se benéfico e essencial para o melhor aproveitamento da GPU. Matrizes pequenas sofrem menos com latência de memória e beneficiam-se mais de otimizações como *shared memory*, enquanto matrizes grandes permitem que milhares de threads se mantenham ocupadas durante longos períodos computacionais.

À medida que n cresce:

- O speedup se estabiliza entre 14× e 16×,
- O kernel básico aproxima-se do desempenho do kernel otimizado,
- A eficiência global aumenta ligeiramente, já que o custo fixo de inicialização e cópia dos dados torna-se menos relevante.

Apesar disso, a eficiência normalizada continua baixa (1–6%), o que é esperado em algoritmos altamente massivos, onde a métrica de eficiência tradicional não reflete bem o comportamento de GPUs modernas.

4.4 Gargalos e Limitações Identificados

Durante os experimentos, alguns fatores foram identificados como principais limites ao desempenho:

4.4.1 Latência da memória global

Embora a GPU ofereça paralelismo massivo, seu desempenho pode ser limitado pela largura de banda da memória global. Para matrizes grandes, tanto o kernel básico quanto o otimizado passam a ser bound pela mesma limitação, reduzindo o impacto da *shared memory*.

4.4.2 Tamanho fixo do bloco

A escolha de blocos quadrados (ex.: 16×16) funciona bem para a maioria dos casos, mas não é necessariamente o melhor para todas as arquiteturas. Dimensões maiores podem explorar melhor os SMs, mas também aumentam o custo de sincronização entre threads.

4.4.3 Custo de cópia CPU -> GPU

Embora esse custo não esteja incluído no cálculo do tempo de kernel, ele ainda representa uma limitação prática quando se considera a execução completa do programa. Para matrizes pequenas, esse overhead pode até superar o tempo do próprio kernel.

4.4.4 Shared memory subutilizada em matrizes grandes

À medida que a matriz cresce, o número de blocos aumenta, mas cada bloco opera apenas sobre pequenos pedaços da matriz. Isso diminui a vantagem do cache explícito, tornando o kernel otimizado apenas marginalmente melhor que o básico.

4.5 Comparação com OpenMP (Projeto 1) e MPI (Projeto 2)

A comparação entre as três abordagens de paralelização: OpenMP, MPI e CUDA, evidencia diferenças marcantes de desempenho e escalabilidade na multiplicação de matrizes DGEMM.

OpenMP apresentou boa escalabilidade em memória compartilhada, alcançando speedups entre 3× e 6×, com desempenho crescente até cerca de 8 threads. Para matrizes grandes (4096×4096), o speedup máximo foi 5.39×, limitado principalmente pela largura de banda da memória e pelo overhead de sincronização entre threads.

MPI, executado em um ambiente de nó único, teve desempenho inferior ao OpenMP devido ao custo de comunicação entre processos. Os speedups ficaram entre 1.2× e 3.7×, melhorando à medida que o tamanho da matriz aumentou, mas com eficiência

reduzida para 8 processos. A ausência de uma rede real e a competição pela mesma memória RAM tornaram a comunicação o principal gargalo.

A implementação em CUDA superou amplamente ambas as abordagens em todos os cenários. Os speedups variaram entre 12.9× e 19.2× para 512×512 e permaneceram estáveis entre 14× e 15.5× para matrizes maiores. Isso se deve ao paralelismo massivo da GPU, ao uso eficiente da memória compartilhada e à capacidade de ocultar latências. Mesmo quando as matrizes aumentam de tamanho, o desempenho da GPU se mantém significativamente superior.

De forma geral, os resultados mostram que:

- CUDA é a abordagem mais eficiente, com speedup 3× maior que OpenMP e até 6× maior que MPI no pior caso para CPU.
- OpenMP é competitivo para máquinas multicore, com boa relação esforço/ganho.
- MPI é limitado no ambiente utilizado, mas ainda demonstra escalabilidade moderada para problemas grandes.

Critério	**OpenMP**	**MPI**	**CUDA**
Melhor Speedup	5.39×	3.78×	19.2×
Speedup Médio (matrizes grandes)	~4–5×	~2.5–3.5×	~14–15×
Eficiência Máxima	0.99	0.87	0.060675(básico) 0.018902(shared)
Escalabilidade	Boa até ~8 threads	Moderada, limitada pela comunicação	Muito alta, limitada pelo hardware da GPU

4.6 Sugestões de Melhorias

Algumas estratégias podem ser aplicadas para ampliar o desempenho e explorar melhor a arquitetura da GPU:

- **Uso de técnicas mais avançadas de tiling**

  Implementar *warp tiling* ou particionamento hierárquico pode reduzir ainda mais acessos à memória global.

- **Aproveitamento de Tensor Cores**

  Para GPUs compatíveis, implementar DGEMM com WMMA (Warp Matrix Multiply Accumulate) poderia multiplicar o desempenho em matrizes grandes.

- **Overlapping de transferência com computação**

  Utilizar *streams* CUDA para sobrepor transferência de dados com execução do kernel pode reduzir o custo total quando várias multiplicações forem realizadas em sequência.

- **Dimensionamento adaptativo dos blocos**

  Autoajustar o tamanho de blocos e grids conforme a arquitetura da GPU pode melhorar a ocupação dos SMs.

- **Benchmarking completo incluindo transferências**

  Medir o tempo total (inclusive cópias) daria uma visão mais realista da aplicação em cenários reais

5. Conclusão

A implementação da multiplicação de matrizes de precisão dupla (DGEMM) em GPU utilizando CUDA permitiu compreender de forma prática como o paralelismo massivo e a hierarquia de memória influenciam diretamente o desempenho em aplicações científicas. Diferentemente do processamento sequencial em CPU, a GPU exige que o desenvolvedor organize a computação em blocos de threads, explorando estratégias como *tiling* e uso eficiente da memória compartilhada para reduzir acessos à memória global.

O principal aprendizado foi perceber como a divisão da matriz em blocos e o carregamento colaborativo de dados pelo bloco elevam drasticamente a performance, diminuindo a latência e aumentando o reaproveitamento de dados. A comparação entre a versão sequencial e os kernels CUDA evidenciou ganhos significativos de velocidade, mostrando que a GPU alcança seu melhor desempenho quando há volume computacional suficiente para saturar os multiprocessadores e amortizar custos de transferência entre CPU e GPU.

Em termos gerais, o projeto consolidou o entendimento sobre organização de threads, limites de registradores, sincronização interna do bloco e o impacto da hierarquia de memória no throughput final da aplicação. Os resultados obtidos confirmam que, para matrizes grandes, a GPU oferece um aumento substancial de desempenho, especialmente quando técnicas de otimização como *shared memory tiling* são aplicadas corretamente.

6. Referências

- NVIDIA. *CUDA Toolkit Documentation*. Disponível em: https://docs.nvidia.com/cuda. Acesso em: 27 nov. 2025.
- ORÉLLANA, E. T. *Repositório Oficial da Disciplina DEC107 – Programação Paralela (2025/2)*. GitHub, 2025. Disponível em: https://github.com/etvorellana/RepoDEC107-PP-2025_2. Acesso em: 27 nov. 2025.
- KIRK, David B.; HWU, Wen-mei W. *Programming Massively Parallel Processors: A Hands-on Approach*. 3. ed. Burlington: Morgan Kaufmann, 2016.