9 A 13 DE SETEMBRO DE 2025
CÂMPUS CEDETEG - UNICENTRO
COMPUTAÇÃO NA ERA DOS DADOS: CARREIRA E OPORTUNIDADES

Estudo e implementação de estratégias para paralelização de convoluções 2D em processadores gráficos (GPUs)

Lucas Machado Cogrossi[1], Josiel Neumann Kuk[1]
contact@lucascogrossi.com, josielkuk@gmail.com

[1]Universidade Estadual do Centro-Oeste (Unicentro)

Resumo:

Este trabalho apresenta o estudo e a implementação de estratégias para paralelização da operação de convolução 2D em GPUs utilizando CUDA. Foram comparadas três versões: sequencial em CPU, paralela ingênua e paralela otimizada. Os experimentos, realizados com matrizes de até 16384x16384 elementos, mostraram ganhos expressivos de desempenho, alcançando speedup superior a 1300 vezes em relação à versão sequencial. Os resultados confirmam a eficiência do modelo de programação paralela em CUDA e sua relevância para aplicações de alto desempenho.

Palavras-chave: Convolução; Computação Paralela; CUDA

1. Introdução

Convolução é uma operação aplicada sobre um conjunto de dados, amplamente aplicada em processamento de imagens, visão computacional e redes neurais convolucionais (CNNs). Em duas dimensões (2D), essa operação é comumente utilizada para extrair características de imagens, como bordas ou texturas. Esta extração é realizada através de um kernel (ou máscara), que é uma matriz de pesos que percorre a imagem, definindo quais características serão extraídas. Nesse contexto, uma convolução 2D é dada por:

$$O_{i,j} = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} K_{u,v} \cdot I_{(i+u),(j+v)}$$

onde I é a imagem de entrada, K a máscara, e O a imagem resultante.

Devido à alta complexidade computacional dessa operação, a execução eficiente dessas operações é essencial para reduzir o tempo de processamento. Nesse contexto, a paralelização deste algoritmo em processadores gráficos (GPUs - *Graphics Processing Unit*) oferece grande vantagem, acelerando significativamente as operações e possibilitando a aplicação em problemas reais (HWUI; KIRK; EL HAJJ, 2023). Neste trabalho, comparamos e implementamos diferentes abordagens para a paralelização de convoluções bidimensionais em GPUs.

2. Materiais e Métodos

O estudo foi realizado em um ambiente computacional com processador AMD Ryzen 7 5700X3D de 8 núcleos e 16 threads, 16 GB de memória RAM e placa gráfica GeForce GTX 1660 Super com 6 GB de memória dedicada e 1408 CUDA Cores. O desenvolvimento foi realizado utilizando WSL (Windows Subsystem for Linux) com Ubuntu 24.04.3 LTS, drivers NVIDIA versão 580.88 e CUDA Toolkit versão 13.0. O codigo fonte está disponível em: https://github.com/lucascogrossi/paralelizacao-convolucao-2d

As implementações foram desenvolvidas em C para a versão sequencial (CPU) e CUDA C/C++ para as versões paralelas em GPU. Para simular imagens, foram utilizadas matrizes de números em ponto flutuante gerados aleatoriamente, com cinco tamanhos distintos: 1024×1024, 2048×2048, 4096×4096, 8192×8192 e 16384×16384 elementos. A máscara de convolução utilizada foi fixada em tamanho 5×5 e inicializada com valores unitários (1.0). Foram implementadas três versões do algoritmo de convolução:

a) Versão sequencial: Implementação tradicional da convolução 2D utilizando loops aninhados para percorrer a matriz de entrada e aplicar a máscara de convolução. Esta implementação utiliza apenas uma thread.

b) Versão paralela ingênua (*naive*)[1]: Primeira implementação paralela. A estratégia consiste na atribuição de uma thread para calcular cada elemento de saída, iterando sobre os elementos de entrada e os pesos da máscara. Utiliza memória constante para armazenamento da máscara. A memória constante é um pequeno espaço de memória que é armazenada em cache e funciona somente para leitura (MAO, 2023). Esta memória é particularmente benéfica pois threads processando pixels adjacentes frequentemente acessam os mesmos coeficientes da máscara, resultando em coalescência de acessos e melhor desempenho.

c) Versão paralela otimizada: Segunda implementação paralela que utiliza a técnica de *tiling* para otimizar o acesso à memória. Nessa versão, a matriz de entrada é dividida em blocos menores (*tiles*) que são carregados em uma memória de menor latência. Cada bloco de threads carrega cooperativamente um tile da matriz de entrada para a memória compartilhada (*shared memory*), permitindo que múltiplas threads reutilizem os mesmos dados sem acessar repetidamente a memória global.

3. Resultados e Discussão

Os resultados são apresentados nas figuras abaixo. A Figura 1 apresenta os resultados para os tempos de execução em relação ao tamanho da matriz e Figura 2 apresenta a correlação entre speedup e tamanho da matriz.

[1] Refere-se a uma implementação simples, direta e não otimizada, muitas vezes desconsiderando aspectos mais complexos ou específicos do hardware.

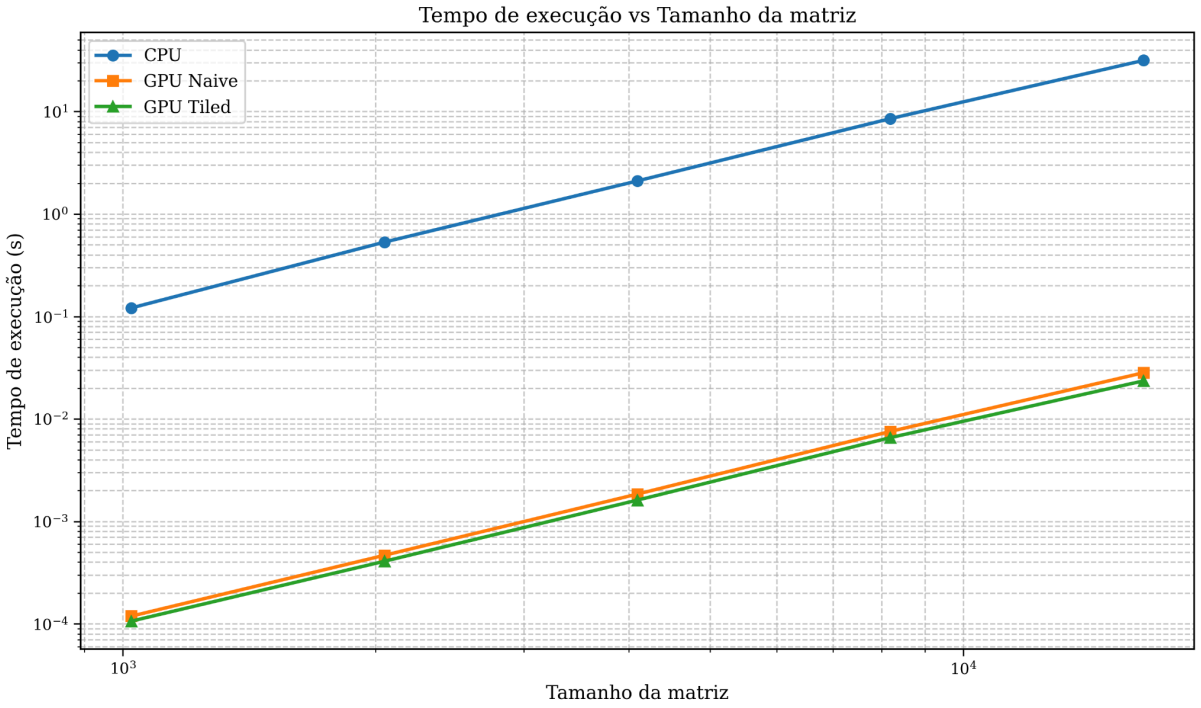

Figura 1: Comparação dos tempos de execução entre CPU e implementações GPU. Fonte: autor

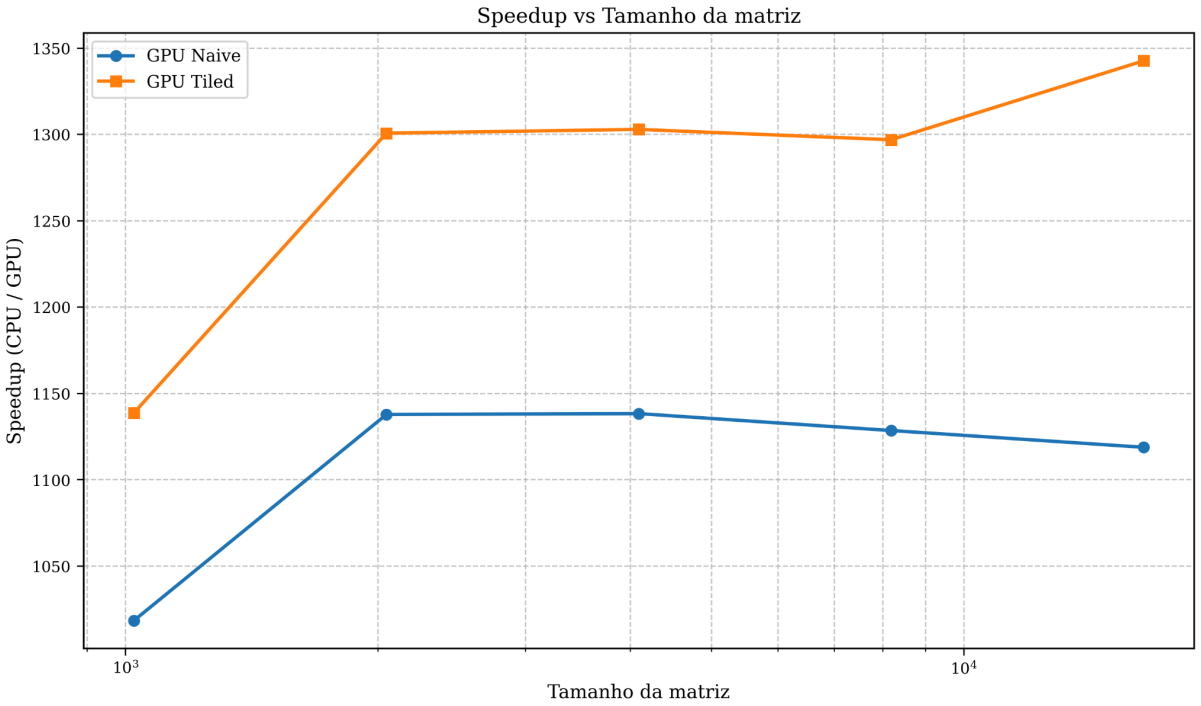

Figura 2: Speedup obtido pelas implementações conforme aumento do tamanho da matriz. Fonte: autor.

A Tabela 1 apresenta os resultados de tempo de execução das três variações. É possível observar que, conforme se aumenta a complexidade do problema, maior é a diferença entre as versões, sendo muito benéfico o uso de paralelismo. O valor de speedup[2] obtido comprova a vantagem de se utilizar uma versão paralela em relação a versão sequencial. Entre as versões paralelas, a versão otimizada obteve melhor desempenho em relação a versão ingênua.

Tabela 1. Comparação de desempenho entre implementações serial e paralelas.

Tamanho matriz	Tempo CPU(s)	Tempo GPU ingênua(s)	Tempo GPU em blocos(s)	Speedup GPU ingênua	Speedup GPU em blocos
1024x1024	0.121088	0.00011889	0.000106330	1018.4	1138.7
2048x2048	0.530868	0.00046660	0.000408110	1137.7	1300.6
4096x4096	2.101968	0.00184720	0.001613400	1138.2	1302.8
8192x8192	8.524190	0.00755430	0.006573000	1128.4	1296.8
16384x16384	31.559052	0.02821100	0.023512000	1118.7	1342.5

4. Considerações finais

Com base nas análises realizadas, os resultados demonstram o sucesso da paralelização da operação de convolução utilizando processadores gráficos. Apesar da complexidade da programação paralela em CUDA, os resultados evidenciam uma grande vantagem nesse modelo de programação. Agradecemos à UNICENTRO e ao Departamento de Ciência da Computação (DECOMP) pela oportunidade de disseminar os resultados desse trabalho.

Referências

[1] HWUI, W. W.; KIRK, D. B.; EL HAJJ, I. **Programming Massively Parallel Processors: A Hands-on Approach**, 4th ed., Morgan Kaufmann, Cambridge, 2023.

[2] MAO, L. **CUDA Constant Memory**. Disponível em: https://leimao.github.io/blog/CUDA-Constant-Memory/ - Acesso em: [17/08/2025].

[2] Métrica que mede o ganho de desempenho obtido ao executar um programa em paralelo, comparado à sua execução sequencial.