Otimização na Simulação e Análise de Modelos de Difusão de Contaminantes em Água

Pedro Paulo F. M. Vianna, Victor Jorge C. Chaves, João Guilherme Iwo D. L. Costa

[1]Instituto Ciência e Tecnologia (ICT) – Universidade Federal de São Paulo (UNIFESP)
Campus São José dos Campos

Abstract. *This work aims to create a simulation that models the diffusion of contaminants in a body of water (such as a lake or river), applying concepts of parallelism to speed up the calculation and observe the behavior of pollutants over time. The project will investigate the impact of OpenMP, CUDA and MPI on model runtime and accuracy. In this document we will address the implementation in OpenMP, parallelizing the code provided in the project proposal.*

Resumo. *Este trabalho tem como objetivo criar uma simulação que modele a difusão de contaminantes em um corpo d'água (como um lago ou rio), aplicando conceitos de paralelismo para acelerar o cálculo e observar o comportamento de poluentes ao longo do tempo. O projeto investigará o impacto de OpenMP, CUDA e MPI no tempo de execução e na precisão do modelo.*

1. Introdução

A difusão de contaminantes em corpos d'água, como lagos, rios e oceanos, é um tema de grande relevância ambiental e social, dada a crescente preocupação com a qualidade da água e o impacto de atividades humanas no meio ambiente. A contaminação por substâncias químicas, resíduos industriais, efluentes domésticos e outros poluentes pode comprometer ecossistemas aquáticos e a saúde pública. Assim, entender e prever a propagação desses contaminantes é fundamental para a formulação de políticas de controle e mitigação.

A partir de simulações computacionais pode ser feito o estudo no comportamento de difusão de poluentes na água. As simulações permitem a análise de diferentes cenários e variáveis, oferecendo uma visão detalhada sobre o comportamento dos poluentes ao longo do tempo e em diferentes condições ambientais. Essas ferramentas proporcionam aos pesquisadores a capacidade de modelar e prever eventos de contaminação.

No entanto, pode ser alta o custo computacional ao simular processos de difusão em grande escala. Para superar esses desafios, métodos de otimização do algoritmo são necessárias para agilizar o tempo de execução das simulações, como também ser capaz de computar casos mais demandantes. Tecnologias como OpenMP (Open Multi-Processing), CUDA (Compute Unified Device Architecture) e MPI (Message Passing Interface) oferecem soluções poderosas para acelerar os cálculos, permitindo o processamento simultâneo de múltiplas operações e a distribuição de tarefas entre diferentes unidades de processamento.

Este estudo se concentra na implementação e comparação dessas técnicas de otimização em simulações de difusão de contaminantes, avaliando seu impacto no desempenho e na precisão dos modelos. Através da utilização de OpenMP, CUDA e MPI, será buscado a

redução do tempo de execução de um algoritmo existente de simulação do processo de difusão em corpos d'água.

2. Descrição do Problema

O objetivo deste estudo é otimizar uma simulação de difusão de contaminantes em corpos d'água, como lagos e rios, utilizando técnicas avançadas de paralelismo e otimização de código. A simulação visa não apenas prever o comportamento dos poluentes ao longo do tempo, mas também avaliar o desempenho computacional de diferentes abordagens de paralelização.

A função que descreve a simulação está detalhada logo abaixo:

$$\frac{\partial C}{\partial t} = D \cdot (\nabla)^2 C$$

Onde:

- C é a concentração do contaminante,
- t é o tempo,
- D é o coeficiente de difusão, e
- $(\nabla)^2 C$ representa a taxa de variação de concentração no espaço.

Este modelo descreve a difusão de um contaminante em um meio homogêneo e isotrópico. A equação baseia-se na Segunda Lei de Fick, que afirma que a taxa de variação temporal da concentração em um ponto é proporcional à divergência do fluxo de difusão nesse ponto.

Para resolver esta equação, utiliza-se o método de diferenças finitas, discretizando o espaço e o tempo. A discretização leva à equação de diferença:

$$\frac{C_{i,j}^{n+1} - C_{i,j}^{n}}{\Delta t} = D \cdot \left(\frac{C_{i+1,j}^{n} + C_{i-1,j}^{n} + C_{i,j+1}^{n} + C_{i,j-1}^{n} - 4C_{i,j}^{n}}{\Delta x^2} \right)$$

Onde:

- $C_{i,j}^{n}$ é a concentração no ponto (i, j) no tempo n,
- Δt é o passo temporal, e
- Δx é o tamanho da célula no espaço discreto.

Este esquema permite calcular a evolução temporal da concentração em uma grade 2D, sendo adequado para implementação em sistemas paralelos, como o modelo estudado neste trabalho.

3. Implementação e Teste

3.1. Descrição da Implementação

Com base no problema, será implementado versões otimizadas e/ou paralelizadas do problema proposto, os casos que serão implementados e testados são o que está listado abaixo:

- Otimização do código sequencial com o compilador GCC usando a *flag* de otimização O3.
- Paralelização com de Threads usando a biblioteca OpenMP.
- Paralelização com Processos usando a biblioteca OpenMPI e otimização de compilação.
- Paralelização com GPU usando CUDA.

3.2. Descrição dos Ambientes de Execução

Cada implementação (exceto o de CUDA) foi executada no seguinte ambiente de execução descrito na tabela abaixo:

Table 1. Descrição do Ambiente de Execução

Nome	Notebook IdeaPad
CPU	Ryzen 5 5600g
GPU Integrada	Radeon Vega 7 Graphic
Memória RAM	12 GB RAM

No caso de CUDA, o ambiente de execução está descrito na tabela abaixo:

Table 2. Descrição do Ambiente de Execução

Nome	Google Colab
CPU	CPU Intel Xeon com 2 vCPUs
GPU Integrada	T4
Memória RAM	16 GB RAM

3.3. Descrição dos casos de Teste

Foi escolhido um conjunto de combinações para simular diferentes cenários de complexidade computacional. Cada implementação (exceto o de CUDA) foi testado nas seguintes combinações de casos:

Table 3. Descrição do Ambiente de Execução

Tamanho da Grid	2000, 4000, 6000
Quantidade de Iterações	500, 1000, 500
Quantidade de Threads (Implementação OpenMP)	2, 4, 6, 8, 10, 12
Quantidade de Processos (Implementação OpenMPI)	2, 4, 6, 8, 10, 12

No caso de CUDA, os casos de teste são os descritos na tabela abaixo:

Table 4. Descrição do Ambiente de Execução

Tamanho da Grid	400, 800, 1200, 1600, 2000, 2400
Quantidade de Iterações	400, 600, 800, 200, 1000, 1200, 1400, 1600, 1800, 2000
Tamanho do Bloco	4, 8, 16, 32

3.4. Otimização com Compilador

Foi utilizado do compilador a flag *-03*, da o qual ativa todas as otimização de código agressivas que o compilador GCC fornece.

3.5. Implementação com OpenOMP

Foi utilizado a diretiva `#pragma omp parallel for` para paralelizar o laço principal, distribuindo a execução entre as threads:

Foi testado o uso da diretiva `collapse(2)` para laços aninhados e `schedule(dynamic, 4)` para balancear carga, mas a proposta original mostrou melhor desempenho geral.

Por fim, foi também analisado com diferentes quantidades de Threads para conferir a performance da solução.

E vale comentar, que não foi utilizado a otimização de compilação pois elas são focadas para execução sequencial.

3.6. Implementação com OpenMPI

A implementação com OpenMPI utiliza decomposição de domínio, onde cada processo recebe um subconjunto de linhas da matriz. A comunicação entre processos ocorre via troca de fronteiras, garantindo continuidade nos cálculos. Cada processo executa as atualizações localmente e sincroniza os dados vizinhos.

Além disso, foi utilizado também a otimização de compilação, para aproveitar a execução sequencial de cada processo.

3.7. Implementação com CUDA

O código utiliza CUDA para paralelizar a solução em uma grade. Foi realizado a distribuição de seções da grade entre múltiplas threads, organizadas em blocos. Cada bloco usa memória compartilhada para armazenar dados temporários, reduzindo a necessidade de acessos à memória global.

Toda a execução descrita acima ocorre para cada iteração, então não há necessidade de sincronização em cada execução do kernel, exceto a sincronização na execução de todas as threads, como também o seu carregamento dos dados para a memória do computador.

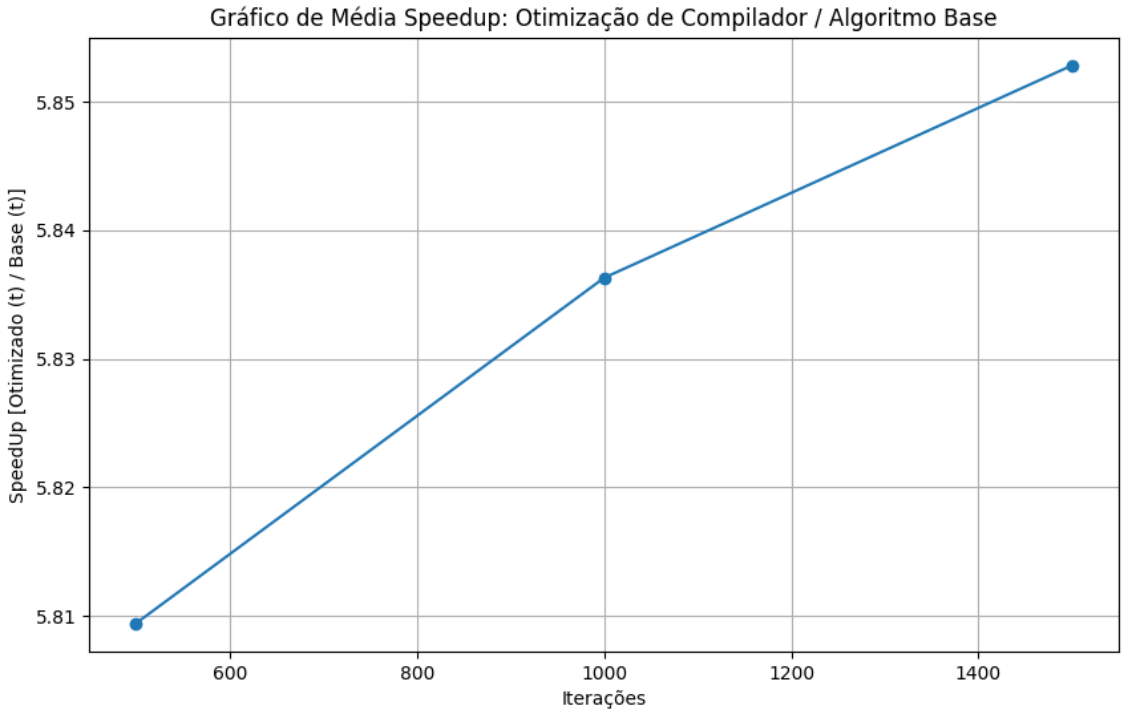
4. Resultados e Discussão

Nesta seção, são apresentados os resultados obtidos para cada técnica de otimização e paralelização aplicada, assim, com os resultados do algoritmo sem otimização, será feito uma análise de speedup para cada técnica.

Para cada técnica, foi conduzida uma série de simulações utilizando combinações variadas de tamanhos de matriz e quantidades de iterações, conforme descrito a seguir:

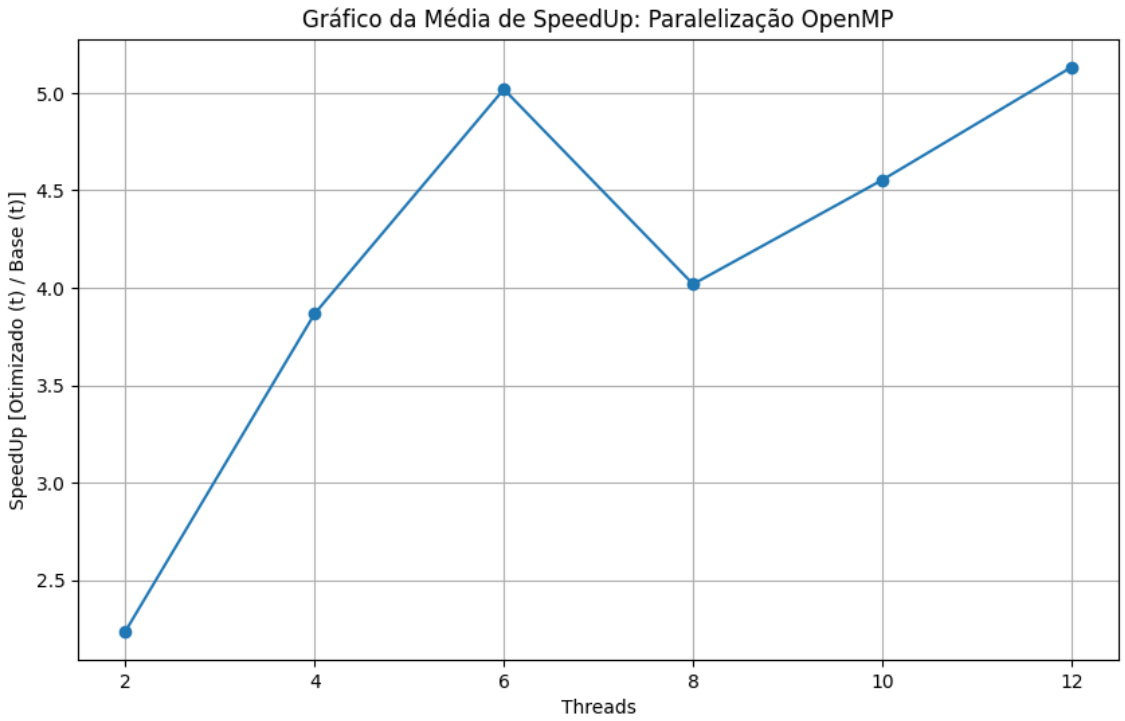
- **Tamanho da Matriz:** 2000, 4000, 6000
- **Quantidade de Iterações:** 500, 1000, 1500

4.1. Resultados do Algoritmo Compilado com a Flag -O3

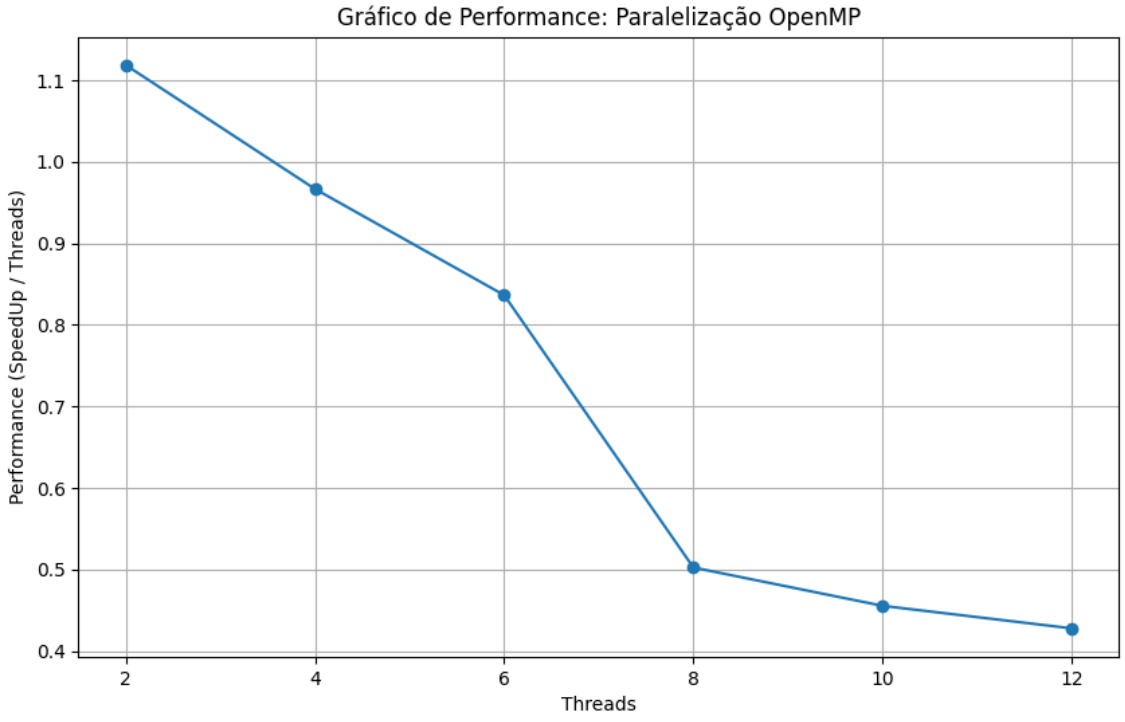


4.2. Resultados do Algoritmo Paralelizado com OpenMP

No gráfico abaixo, sobre a média de SpeedUp, os dois casos com o melhor SpeedUp é com 12 e 6 threads, invés de ser 12 e 10 threads, apesar de não ser encontrado um motivo para isso, pode ser que na configuração de 6 processos tem o menor overhead em comparação ao tempo ganho com a paralelização.

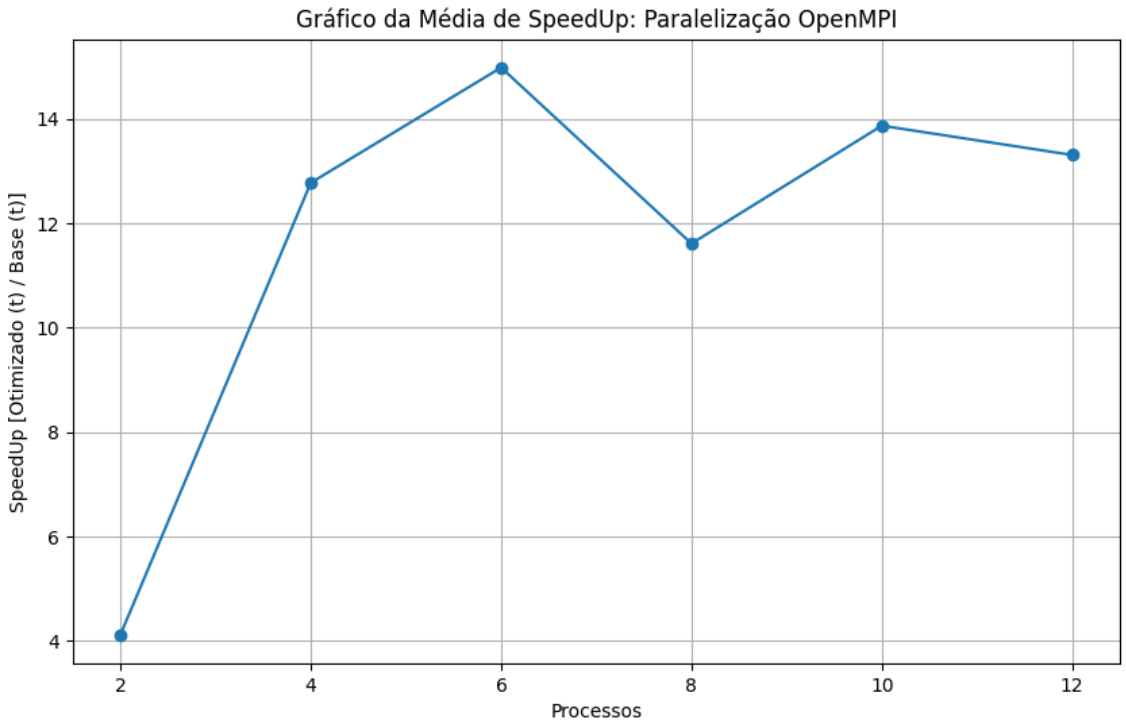


No gráfico abaixo, sobre a performance, é relevante a queda de performance conforme mais threads são usadas, indicando assim que quanto mais threads menos eficiente é o uso das CPUs.

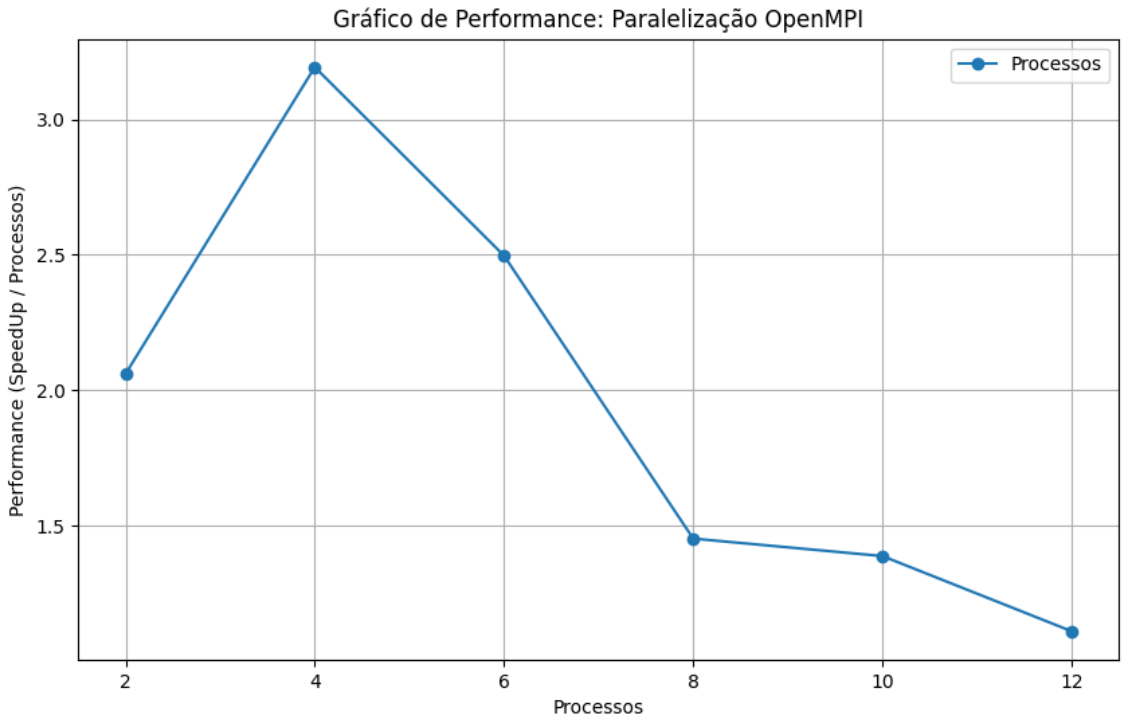


4.3. Resultados do Algoritmo Paralelizado com OpenMPI e com Otimização de Compilação

No gráfico abaixo, sobre a média de SpeedUp, os dois casos com o melhor SpeedUp é com 6 (a melhor) e 10 processos, invés de ser 12 e 10 processos, isso pode ocorrer por diminuir a quantidade de sincronização necessária entre processos enquanto há uma divisão adequada de trabalho entre os processos.



No gráfico abaixo, sobre a performance, é relevante notar que em comparação a paralelização com OpenOMP, a performance com MPI sempre está acima que 1, enquanto do OpenOMP esteve em sua maioria, abaixo de 1, assim indicando uma ótima utilização dos recursos do computador quando a atividade é dividida entre processos que em threads.

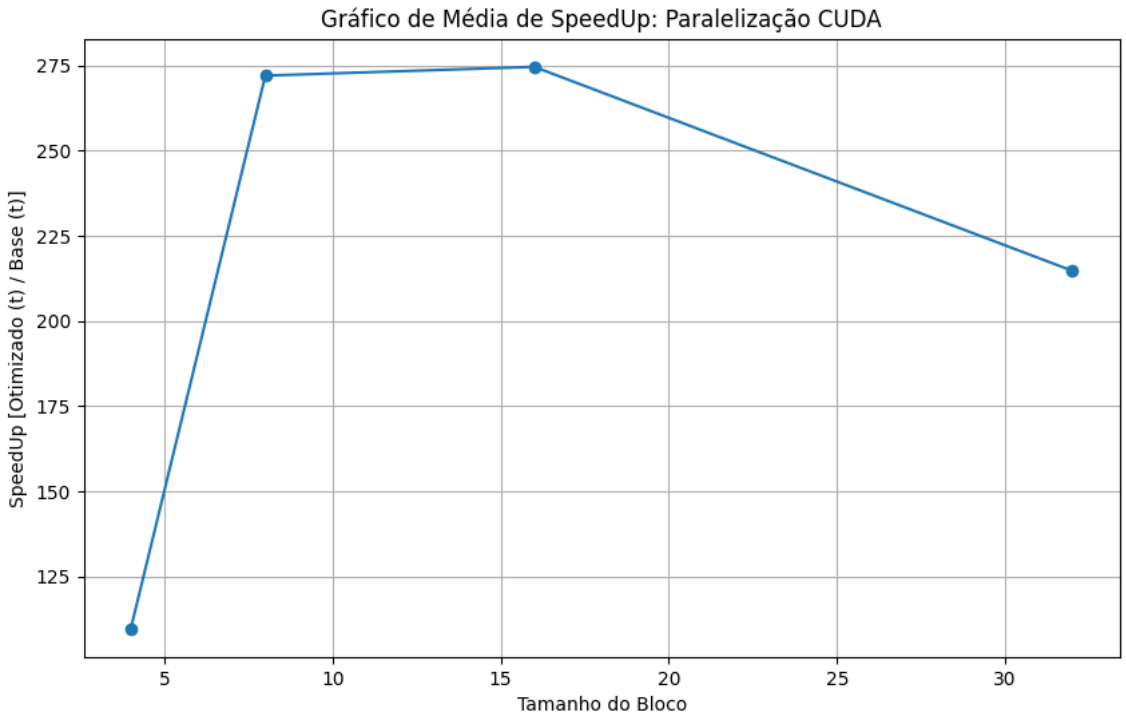


4.4. Resultados do Algoritmo Adaptado para CUDA

4.4.1. Configuração do Ambiente de Execução

- Máquina: Ambiente Virtual do Google Colab
- CPU:
- Memória RAM: 16 GB
- GPU: T4
- Compilador: NVCC (NVIDIA CUDA Compiler)

No gráfico abaixo de SpeedUp, comparando cada caso com um tamanho de bloco diferente, é notável que com 16 de tamanho de bloco é o que apresentou o melhor SpeedUp. Esse comportamento pode ser explicado pelo balanceamento entre a ocupação da GPU e a minimização da sobrecarga associada à troca de contexto entre warps. Blocos muito pequenos podem não explorar eficientemente a capacidade dos multiprocessadores da GPU, enquanto blocos muito grandes podem causar contenção de recursos e maior latência na comunicação entre threads.



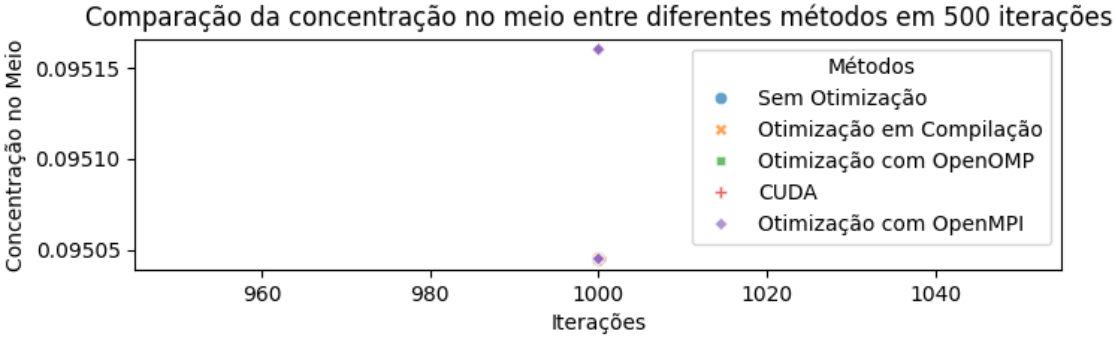
4.5. Resultados da Concentração e de Assertividade

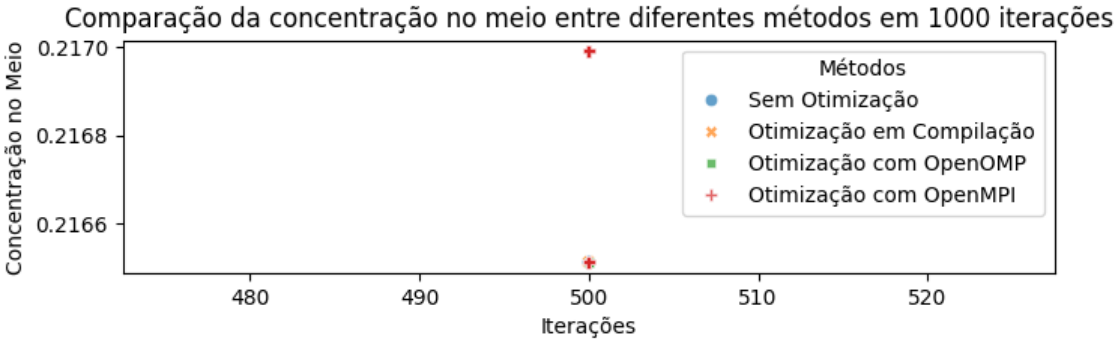
Abaixo, é apresentado os resultados da concentração no meio em qualquer grade da matriz em relação a quantidade de iterações.

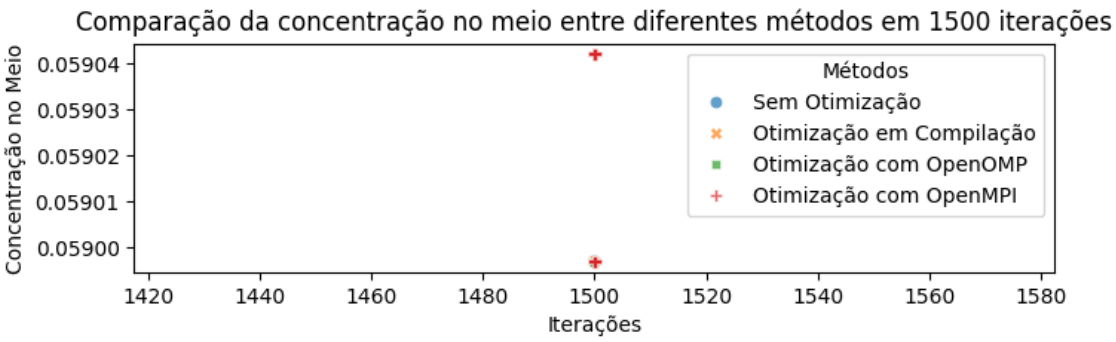
Table 5. Resultados Corretos da Concentração (Com Matriz de Tamanho 2000, 4000 ou 6000)

Iterações	Concentração no Meio
500	0.216512
1000	0.095045
1500	0.058997

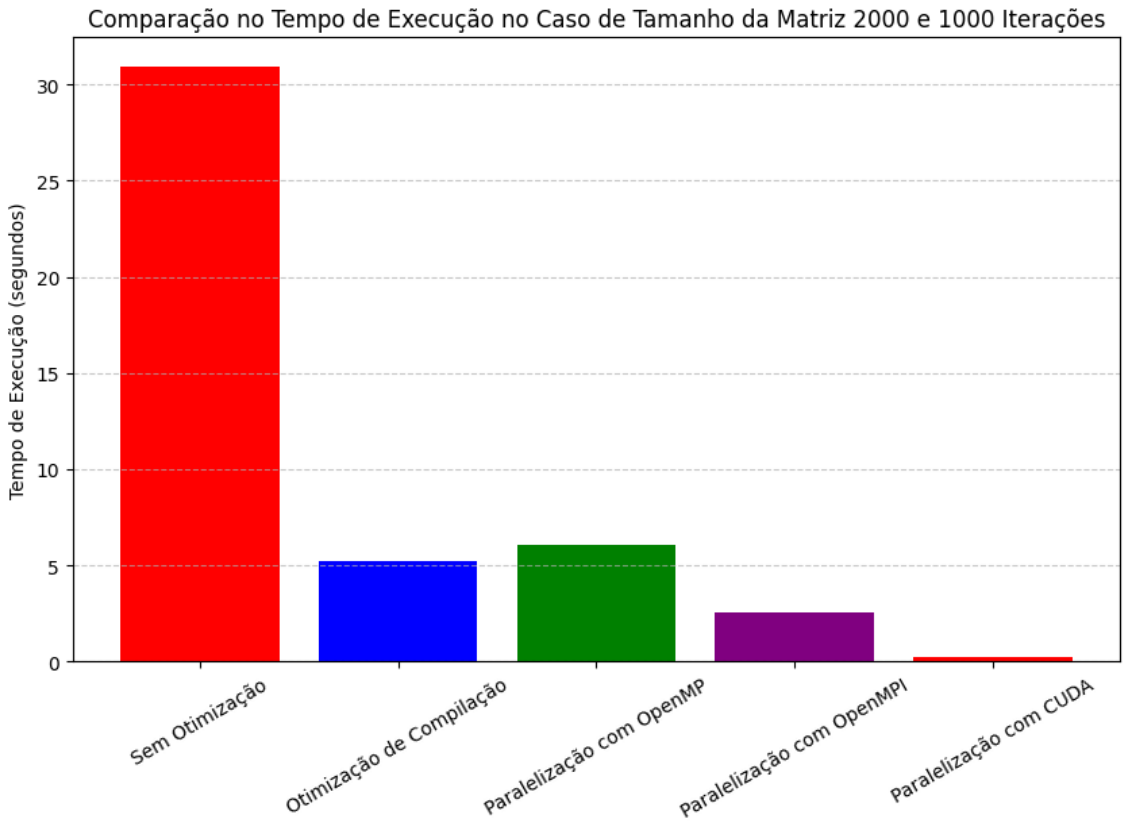
Nos próximos três gráficos, é apresentado em visão de dispersão os resultados alcançados pelos métodos em cada quantidade de iterações, apesar de o algoritmo do CUDA apenas entrar no caso de 1000 iterações, todos eles tem apresentado resultados iguais ao original, todavia, alguns casos com o algoritmo do MPI estiveram fora do padrão na terceira casa do decimal.


Comparação da concentração no meio entre diferentes métodos em 500 iterações
Concentração no Meio
0.09515
0.09510
0.09505
960
980
1000
1020
1040
Iterações
Métodos
Sem Otimização
Otimização em Compilação
Otimização com OpenOMP
CUDA
Otimização com OpenMPI


Comparação da concentração no meio entre diferentes métodos em 1000 iterações
Concentração no Meio
0.2170
0.2168
0.2166
480
490
500
510
520
Iterações
Métodos
Sem Otimização
Otimização em Compilação
Otimização com OpenOMP
Otimização com OpenMPI


Comparação da concentração no meio entre diferentes métodos em 1500 iterações
Concentração no Meio
0.05904
0.05903
0.05902
0.05901
0.05900
1420
1440
1460
1480
1500
1520
1540
1560
1580
Iterações
Métodos
Sem Otimização
Otimização em Compilação
Otimização com OpenOMP
Otimização com OpenMPI

4.6. Comparativo com todos os Métodos



Os resultados obtidos evidenciam que a implementação CUDA apresentou o melhor desempenho, mas houve uma falha na obtenção de mais dados, o que limitou a análise mais aprofundada da escalabilidade da solução.

Entre as otimizações com CPU, OpenMPI apresentou o melhor resultado utilizando de 6 processos. E curiosamente, a versão sequencial otimizada com a flag -O3 do GCC superou a implementação OpenMP com 12 threads. Isso sugere que a otimização agressiva aplicada pelo compilador, foi suficiente para melhorar o desempenho sem a necessidade do overhead de gerenciamento de múltiplas threads, indicando que problemas de itensivo iteração pode ser otimizados em compilação.

Por fim, os resultados dos modelos tem sido extremamente fiéis ao resultado original, todavia, há alguns casos que o algoritmo do MPI tem errado na concentração final no meio da grade, tal erro pode ser por conta de algum problema de sincronização que ocorreu durante a execução.

5. Conclusão

Os resultados indicam que a utilização de GPUs com CUDA se mostra particularmente eficiente para problemas do tipo stencil 2D, alcançando uma execução 275 vezes mais rápido que o algoritmo original. O processamento de células pode ser altamente paralelizado devido à sua natureza estruturada e ao alto grau de reutilização de dados vizinhos. Essa característica permite um melhor aproveitamento da hierarquia de memória da GPU, reduzindo a latência e aumentando a largura de banda efetiva. Trabalhos futuros podem explorar diferentes estratégias de acesso à memória global e compartilhada para melhorar ainda mais a eficiência da computação stencil em arquiteturas CUDA.

Outra alternativa utilizando apenas de CPU é a execução em processos via OpenMPI aproveitando de uma otimização de compilação, com esse conjunto de técnicas é possível reduzir ainda mais o tempo de execução, alcançando um SpeedUp de 15, mas comparando com a execução de GPU, é 18 vezes mais lenta.

6. References

NVIDIA Corporation. *CUDA C Programming Guide*. Santa Clara, CA: NVIDIA, 2024. Disponível em: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Acesso em: 10 fev. 2025.

Carleton University. *Introduction to MPI*. Disponível em: `https://carleton.ca/rcs/rcdc/introduction-to-mpi/`. Acesso em: 10 fev. 2025.

Calvin University. *Matrices*. Disponível em: `https://cs.calvin.edu/courses/cs/112/labs/10/matrices/`. Acesso em: 10 fev. 2025.