

COMPUTAÇÃO PARALELA - CIÊNCIA DA COMPUTAÇÃO

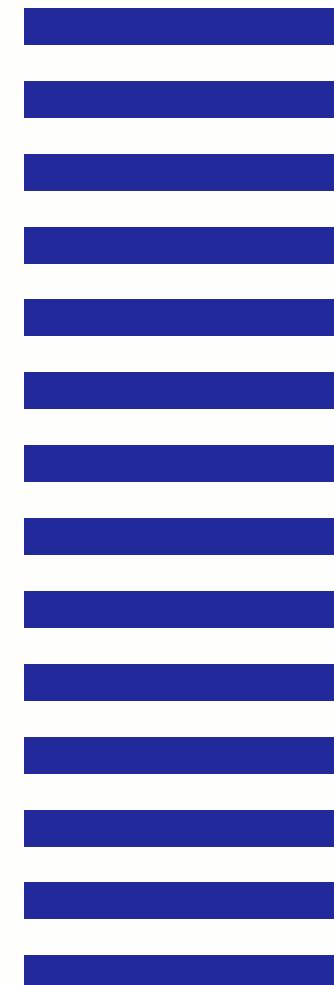
Implementação Paralela do Algoritmo K-Means (OpenMP-GPU/CUDA)

Dã Gonçalves
Hanna Chaves
Laura Caetano
Lucas Sena
Luiz Carmo
Samuel Lopes

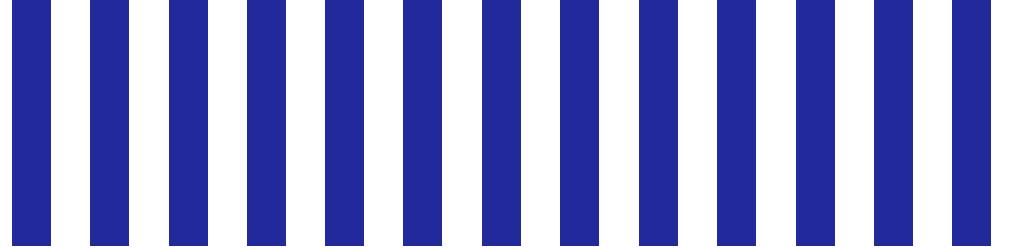
LINK DO REPOSITÓRIO

<https://github.com/da-pet/kmeans-mpi>

SUMÁRIO

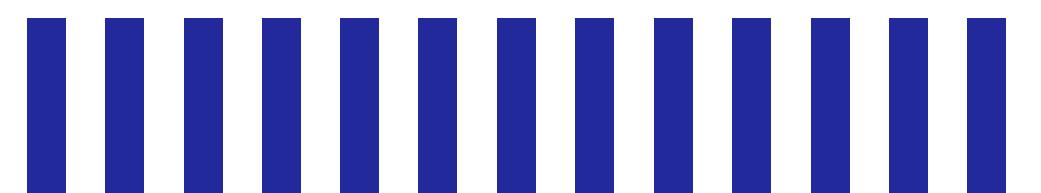


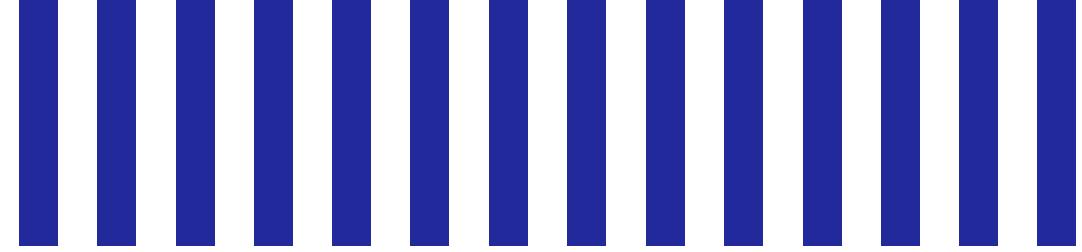
- 01 Introdução
- 02 Base de Dados
- 03 Algortimo de K-means
- 04 Implementação - OpenMP
- 05 Implementação - Cuda
- 06 Resultados - OpenMP
- 07 Resultados - Cuda
- 08 Conclusão



INTRODUÇÃO

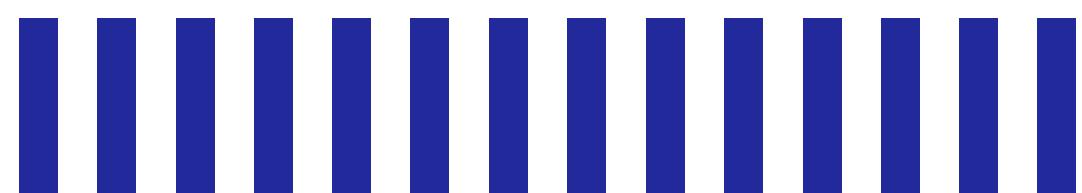
- O trabalho investiga como diferentes modelos de paralelização, OpenMP, OpenMP com offloading para GPU e CUDA, afetam o desempenho do algoritmo K-Means aplicado a um dataset real de grande escala.
- Cada versão foi desenvolvida para aproveitar diferentes níveis de paralelismo, desde múltiplas threads na CPU até execução massiva na GPU.
- O objetivo central é comparar o comportamento real das três estratégias, medindo ocupação, eficiência de warp e tempo final para compreender qual solução entrega melhor desempenho para esse tipo de processamento iterativo.





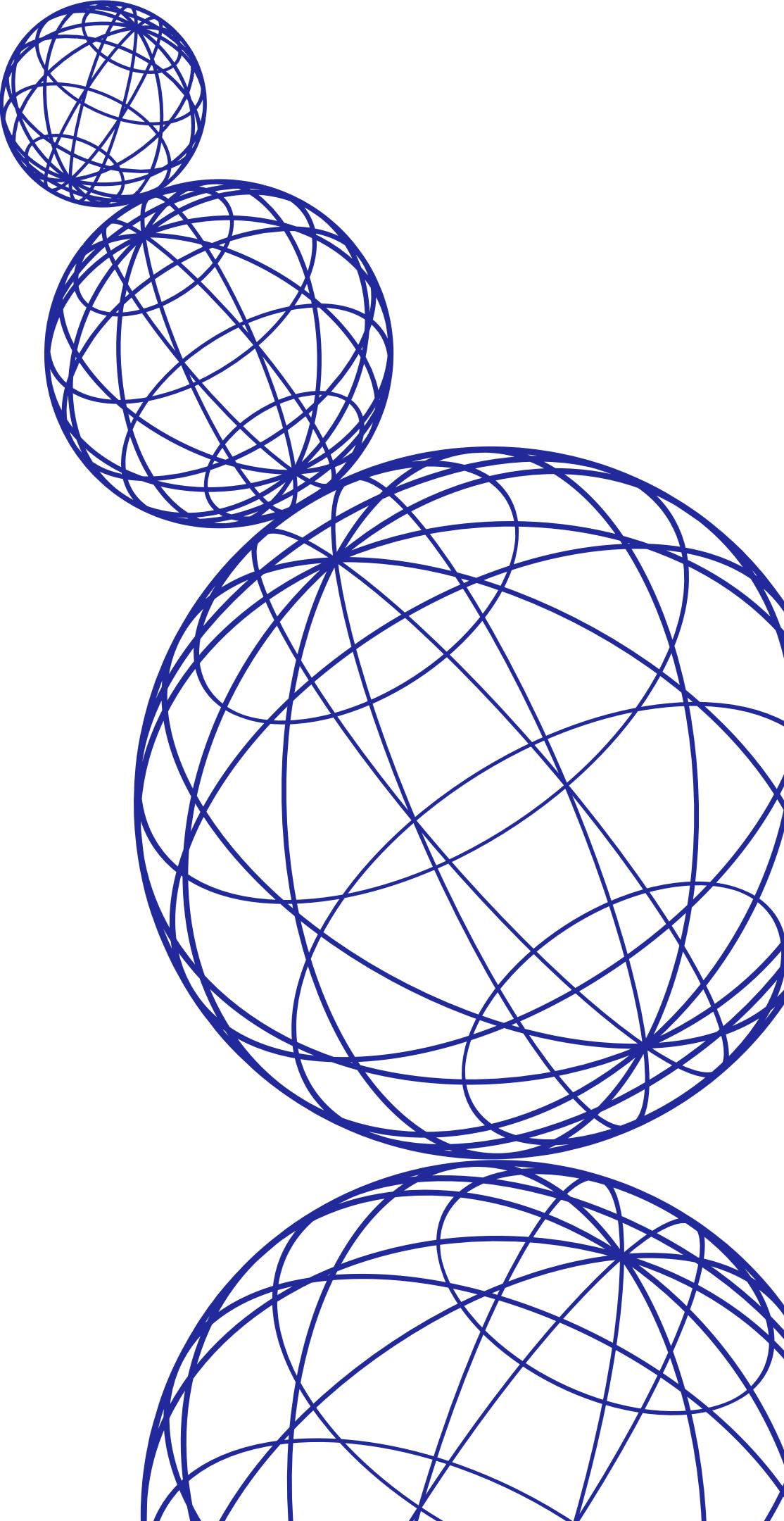
BASE DE DADOS - COVERTYPE

- A base Covertype, do repositório UCI Machine Learning, contém 581.012 amostras e 54 atributos (10 contínuos e 44 binários), descrevendo características ambientais de áreas florestais no Parque Nacional Roosevelt, nos EUA.
- Cada linha representa uma área de 30×30 metros, com informações como elevação, declividade, distância de rios e estradas, tipo de solo e região silvestre — todas em formato numérico, ideal para o algoritmo k-means.
- Escolhemos essa base por ser grande e desafiadora, garantindo tempo de execução acima de 10 segundos, por ser numérica, bem estruturada.
- Além disso ela é amplamente usada em testes de desempenho e benchmarks de aprendizado não supervisionado.



Algoritmo de k-means

- É um algoritmo de agrupamento não supervisionado.
- Agrupa os pontos em K grupos (clusters) com base na semelhança entre eles.
- Cada grupo é representado por um centróide, que é a média dos pontos do grupo.
- Etapas principais:
 1. Escolher K centróides iniciais.
 2. Atribuir cada ponto ao centróide mais próximo.
 3. Recalcular os centróides como a média dos pontos atribuídos.
 4. Repetir até que os centróides não mudem mais (convergência).



Algoritmo de k-means

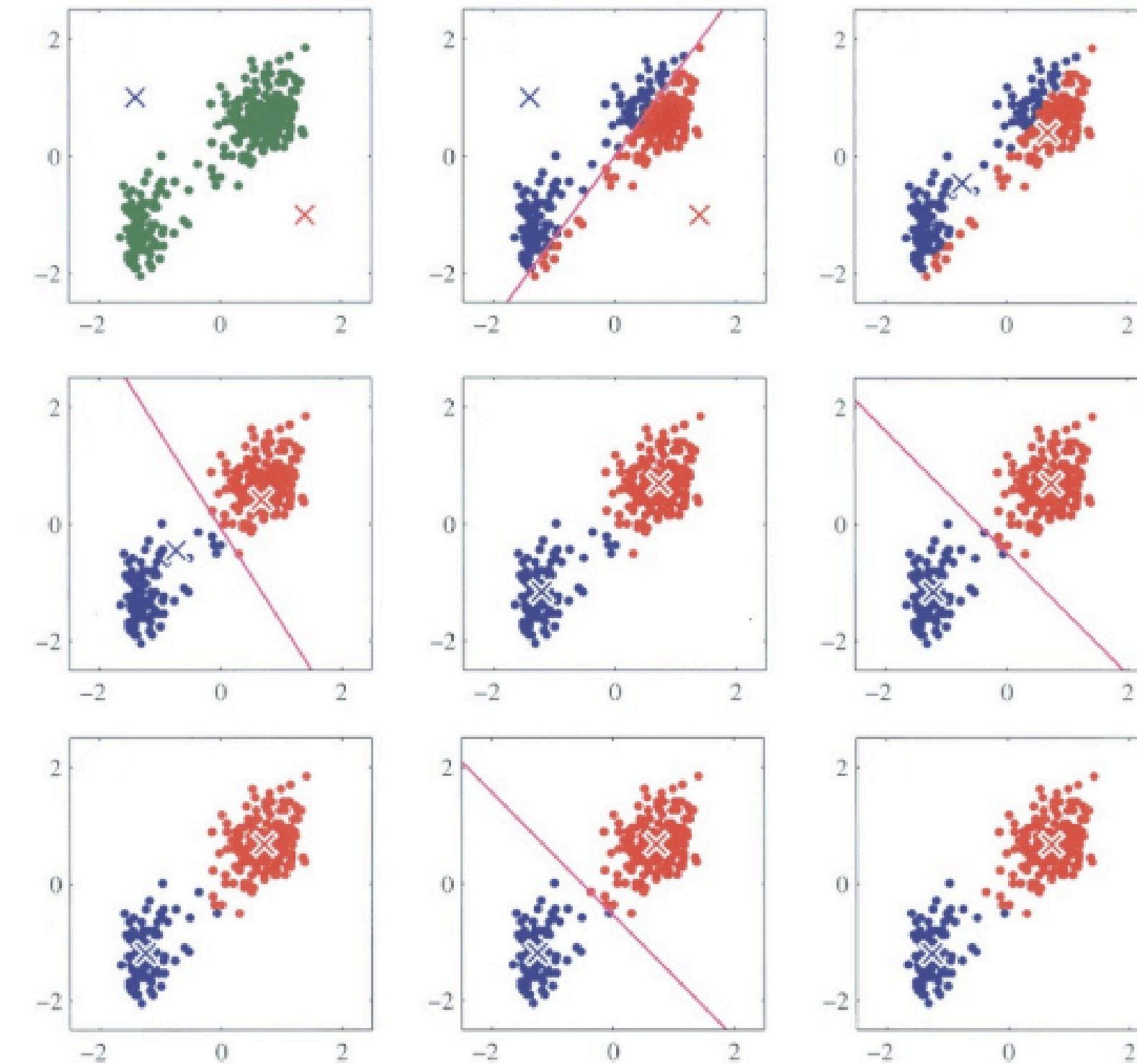


Figure 1 – Processing the Old Faithful Geyser dataset using the k-means algorithm
<https://github.com/KlimentLagrangiewicz/k-means-in-C>

Implementação - OpenMP

- Target Offloading ativado
 - Execução enviada para a GPU usando **#pragma omp target**
 - Verificação automática para saber se o código realmente rodou na GPU
- Dados enviados uma única vez para a GPU
 - Dataset (x) copiados para a memória da GPU
 - Evita custo de transferência a cada iteração
- Atribuição de clusters paralelizada
 - Cada ponto do dataset processado por uma thread na GPU
 - Distância e cluster mais próximo calculados em paralelo
 - **atomic** protege o incremento de **nums[L]**

Implementação - OpenMP

- Reatribuição dos pontos também paralelizada
 - Reavaliação dos pontos após recalcular centróides
 - Verifica mudanças de cluster de forma massiva e paralela
- Cálculo dos centróides mantido na CPU
 - Reduções numéricas (somatórios) feitas no host por estabilidade e compatibilidade
 - Centróides atualizados reenviados para a GPU a cada iteração
- Arquitetura híbrida CPU/GPU
 - GPU executa loops de alto custo computacional
 - CPU executa reduções e controla o fluxo do algoritmo

Código- OpenMP

```
#pragma omp target teams distribute parallel for
for (int i = 0; i < n; ++i) {
    int l = get_cluster(x + (size_t)i * m, c, m, k);
    y[i] = l;
    #pragma omp atomic
    nums[l]++;
}
```

Implementação - CUDA

- Na **CPU**:
 - Escolhe os centróides iniciais
 - Aloca memória global na **GPU** (dataset e centróides)
 - Copia os dados para a memória alocada na **GPU**. Aproximadamente **239.3MB (n * m * 8 bytes)**
- Na **GPU**:
 - Cada thread calcula a distância entre seu ponto e todos os centróides, atribuindo o centróide de menor distância.
 - Retorna para **CPU** o vetor de rótulos e vetor de contagem de pontos por cluster
- Na **CPU**:
 - Os centróides são recalculados usando a média e mandados para a **GPU**

Esse processo acontece até o algoritmo convergir.

Kernels

```
__device__ int get_cluster_gpu(const double *x, const double *c, int m, int k) ...
```

- Recebe vetor de features (ponto) e matriz de centróides
- Thread calcula distância do ponto e dos centróides
- Retorna índice do centróide de menor distância

```
__global__ void kernel_start_partition(const double *d_x, const double *d_c,
                                         int *d_y, int *d_nums, int n, int m, int k)
```

Atribuição inicial:

- Calcula idx da thread
- Chama o kernel `get_cluster_gpu` para o ponto que a thread é responsável
- Escreve no vetor de rótulos qual cluster aquela instância pertence
- Usa `atomicAdd()` para permitir que várias threads escrevam na mesma posição do vetor `d_nums` que guarda número de pontos atribuídos à aquele cluster

Kernels

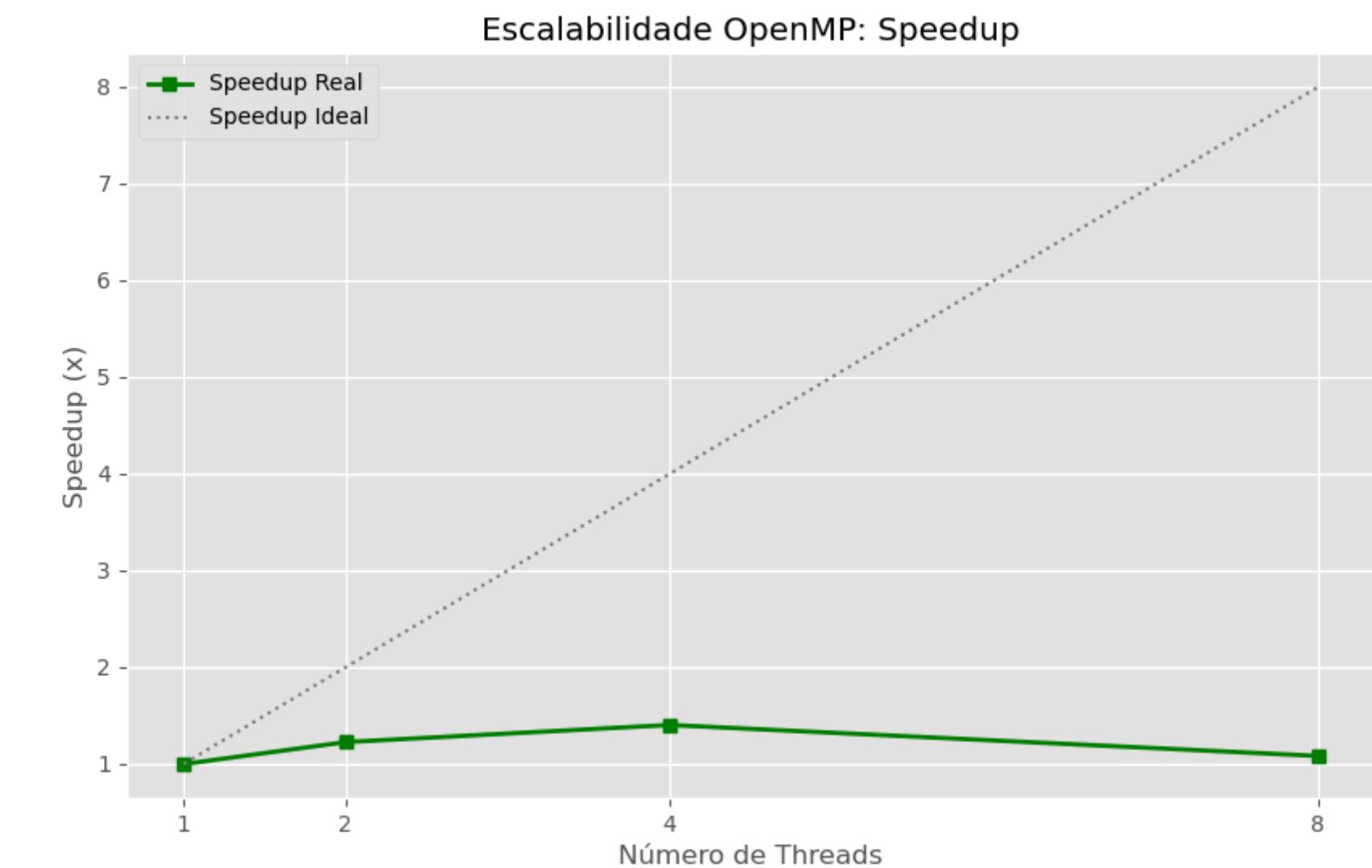
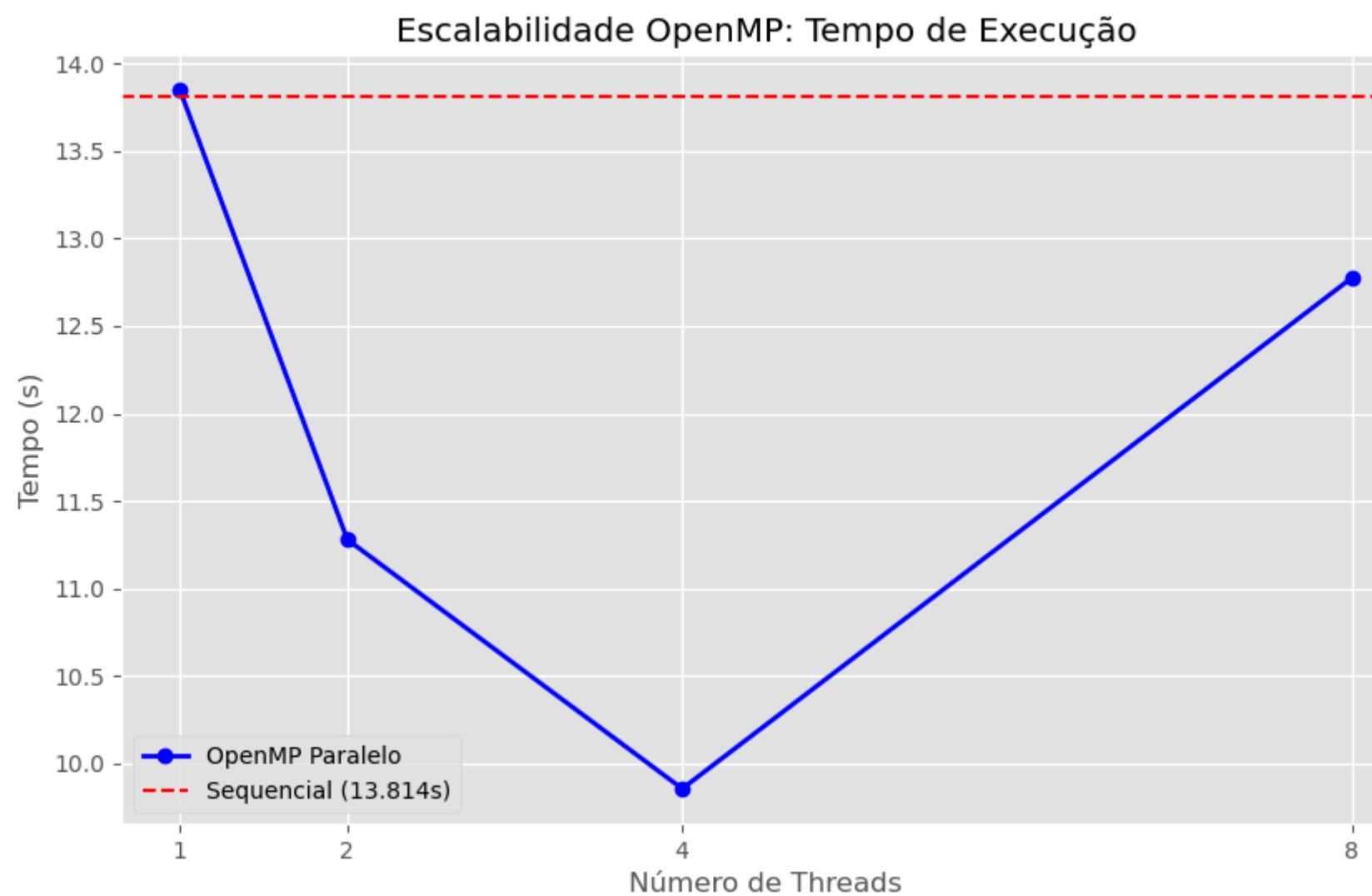
```
__global__ void kernel_check_partition(const double *d_x, const double *d_c,
                                         int *d_y, int *d_nums, int *d_changed, int n, int m, int k)
```

- Calcula idx da thread
- Salva o cluster atual do ponto que a thread está responsável
- Calcula o novo cluster com o kernel `get_cluster_gpu()`
- Verifica se o cluster mudou:
 - Se tiver mudado, significa que o algoritmo ainda não convergiu
 - `atomicExch`: todas as threads que mudaram de cluster, vão sobreescriver a variável `d_changed` com o valor 1, Indicando que ainda devem ser executadas mais iterações.
- Atualiza o vetor de rótulos com o cluster do ponto
- `atomicAdd`: todas as threads que pertencem àquele cluster somam 1 ao vetor de quantidade de pontos por cluster.

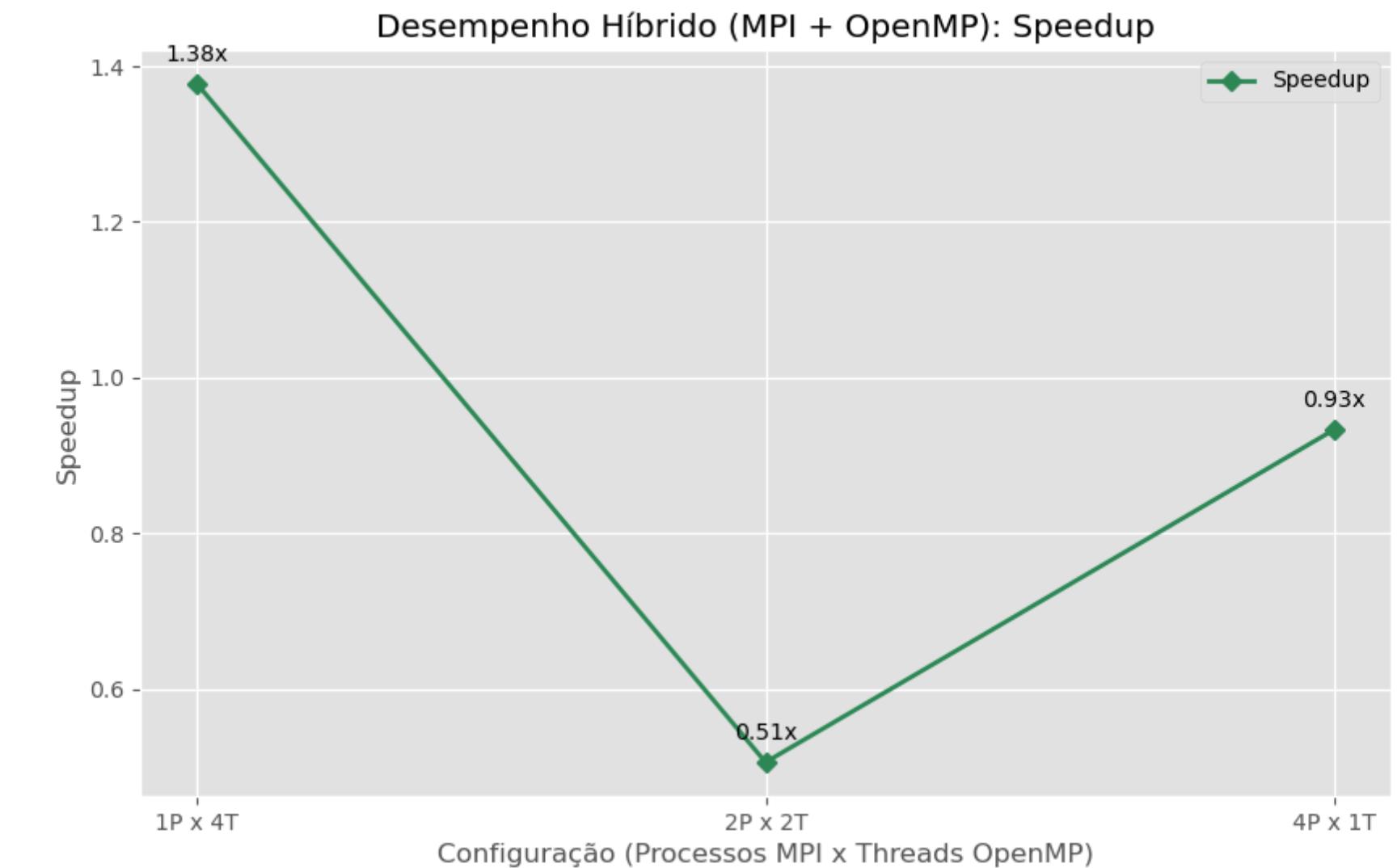
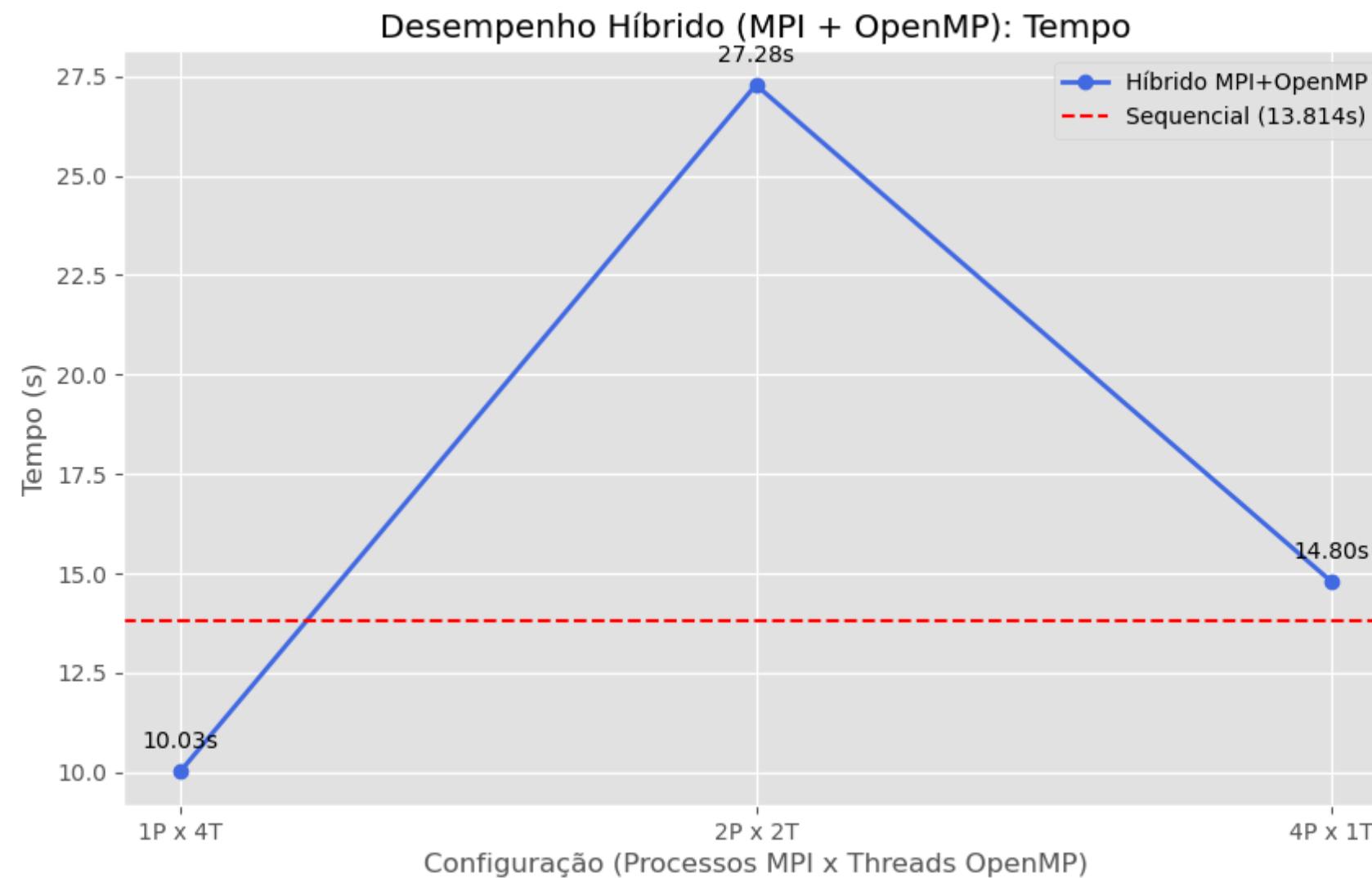
Implementação - CUDA

- GT 1030 (Pascal):
 - 3 SMs (128 CUDA cores por SM)
 - Cada SM possui 4 warp schedulers (até 12 warps por ciclo)
- Bloco: 256 threads → (256,1,1)
- Grid: 2270 blocos → (2270,1,1)

Resultados OpenMP



Resultados MPI+OpenMP



Resultados - OpenMP-GPU

Parcode: GPU Nvidia GT 1030 → 384 núcleos

Metric Description	Min	Max	Avg
Warp Execution Efficiency	99.76%	99.77%	99.77%
Achieved Occupancy	0.249959	0.249996	0.249987
Eligible Warps Per Active Cycle	0.385192	0.386987	0.386059
Issue Stall Reasons (Data Request)	48.14%	48.47%	48.24%
Multiprocessor Activity	99.10%	99.50%	99.31%
Executed IPC	0.405741	0.407871	0.407009
Warp Execution Efficiency	99.76%	99.76%	99.76%
Achieved Occupancy	0.112265	0.112265	0.112265
Eligible Warps Per Active Cycle	0.098118	0.098118	0.098118
Issue Stall Reasons (Data Request)	9.21%	9.21%	9.21%
Multiprocessor Activity	27.38%	27.38%	27.38%
Executed IPC	0.088741	0.088741	0.088741
Warp Execution Efficiency	99.80%	99.80%	99.80%
Achieved Occupancy	0.249991	0.249991	0.249991
Eligible Warps Per Active Cycle	0.398614	0.398614	0.398614
Issue Stall Reasons (Data Request)	46.77%	46.77%	46.77%
Multiprocessor Activity	99.58%	99.58%	99.58%
Executed IPC	0.417211	0.417211	0.417211

```
nvprof --metrics
warp_execution_efficiency,achieved_occupancy,eligible_warps_per_cycle,stall_memory_dependency,sm_activity,executed_ipc
./omp_gpu ./datasets/covertype/data.txt 581012 54 7
./datasets/covertype/new_result.txt ./datasets/covertype/res.txt
```

check_partition()

- Eficiência de Execução da Warp: **99.77%**
- Occupancy Alcançada: **0.249987**
- Dependência de Memória Parada: **48.24%**
- Atividade da SM: **99.31%**
- IPC: **0.407009**

det_start_partition()

- Eficiência de Execução da Warp: **99.80%**
- Occupancy Alcançada: **0.249991**
- Dependência de Memória Parada: **46.77%**
- Atividade da SM: **99.58%**
- IPC: **0.417211**

Tempo Sequencial: 13.814s

Tempo OpenMP: 22.812s

Resultados - Cuda

Parcode: GPU Nvidia GT 1030 → 384 núcleos

Metric Description	Min	Max	Avg
inst *, int*, int*, int, int, int)			
Warp Execution Efficiency	100.00%	100.00%	100.00%
Achieved Occupancy	0.934312	0.934312	0.934312
Eligible Warps Per Active Cycle	0.061206	0.061206	0.061206
Issue Stall Reasons (Data Request)	50.30%	50.30%	50.30%
Instructions Executed	52401126	52401126	52401126
Shared Load Transactions	0	0	0
Shared Store Transactions	0	0	0
Multiprocessor Activity	99.90%	99.90%	99.90%
Executed IPC	0.061471	0.061471	0.061471
inst *, int*, int*, int*, int, int, int)			
Warp Execution Efficiency	99.89%	100.00%	99.97%
Achieved Occupancy	0.877699	0.939352	0.919433
Eligible Warps Per Active Cycle	0.054556	0.056676	0.055605
Issue Stall Reasons (Data Request)	52.12%	55.16%	54.18%
Instructions Executed	50821467	50888577	50840403
Shared Load Transactions	0	0	0
Shared Store Transactions	0	0	0
Multiprocessor Activity	99.82%	99.88%	99.86%
Executed IPC	0.058296	0.060526	0.059004

Tempo sequencial: 13.814s

nvprof --metrics

```
ipc,sm_efficiency,warp_execution_efficiency,achieved_occupancy,eligible_warps_per_
cycle,stall_memory_dependency,inst_executed,shared_load_transactions,shared_sto
re_transactions,global_load_transactions,global_store_transactions ./cudad ./
datasets/covertpe/data.txt 581012 54 7 ./datasets/covertpe/new_result.txt ./
```

Kernel: kernel_start_partition()

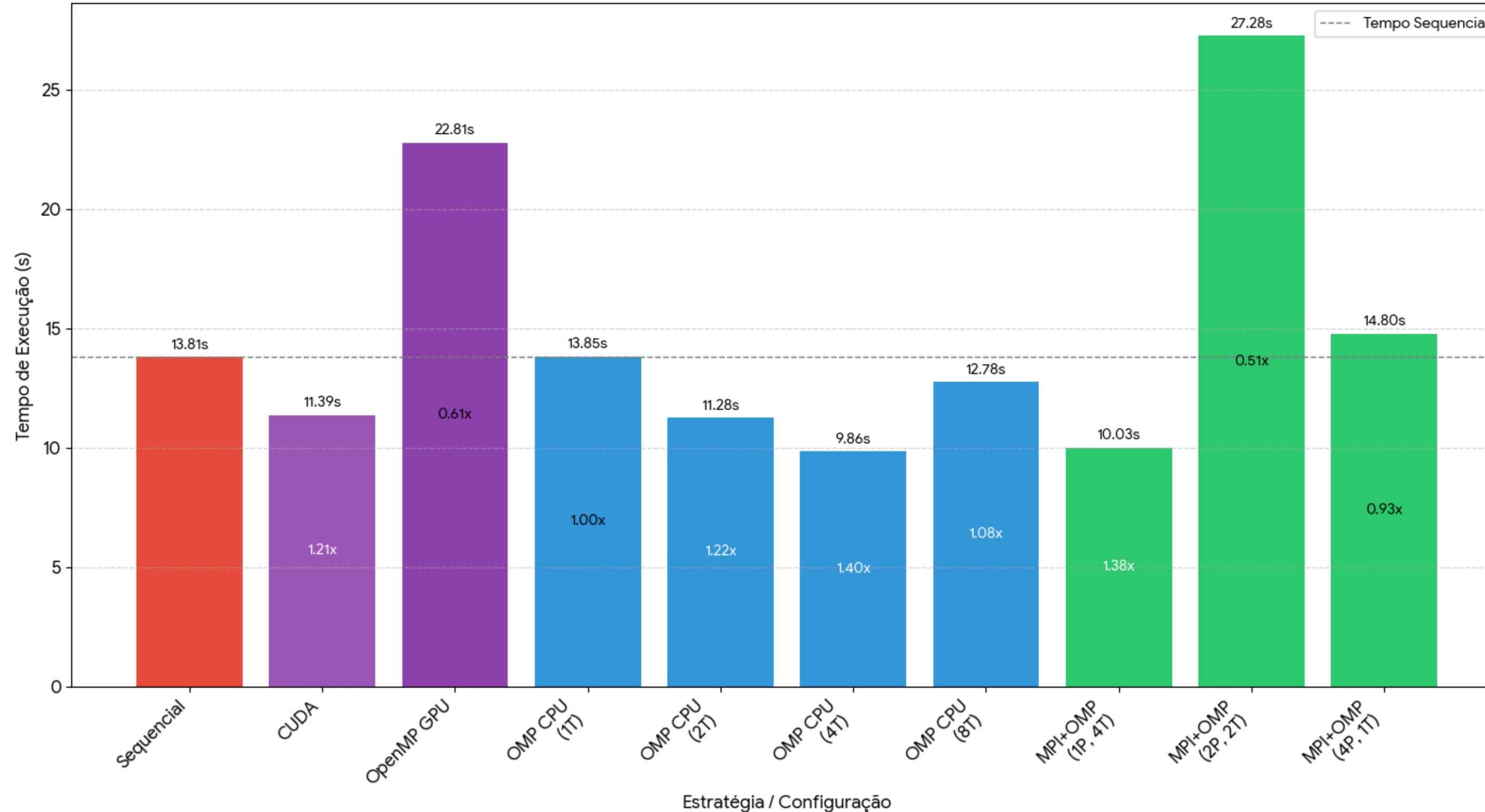
- Eficiência de Execução da Warp: **100.00%**
- Occupancy: **0.934312**
- Atividade da SM: **99.90%**
- IPC: **0.061471**
- Dependência de Memória Parada: **50.30%**
- Warps Elegíveis por Ciclo: **0.061206**
- Num. Instruções: **52401126**

Kernel: kernel_check_partition()

- Eficiência de Execução da Warp: **99.97%**
- Occupancy: **0.919433**
- Atividade da SM: **99.86%**
- IPC: **0.059004**
- Dependência de Memória Parada: **54.18%**
- Warps Elegíveis por Ciclo: **0.055605**
- Num. Instruções: **50840403**

Resultados - Comparações de tempo de Execução

Comparação Completa de Todas as Estratégias e Configurações





Conclusão

- A execução com OpenMP na CPU utilizando 4 threads apresentou o melhor tempo total, mostrando que, para esse problema e tamanho de dataset, o paralelismo moderado na CPU gerou melhor aproveitamento dos recursos do que a execução massivamente paralela na GPU.
- Os experimentos com OpenMP Offloading evidenciaram que a GPU ficou subaproveitada, com occupancy baixa e um custo de transferência de dados alto o suficiente para comprometer o desempenho. Mesmo com boa eficiência de warp, a GPU não compensou o overhead estrutural do offloading.
- A versão CUDA, apesar de contar com kernels dedicados e atingir números mais altos de occupancy e SM activity, foi limitada pela capacidade reduzida da GT 1030, que não possui largura de banda ou número de núcleos suficientes para superar a CPU quando o custo de iterações sucessivas do K-Means é considerado.
- Em conjunto, os resultados mostram que o K-Means não escala automaticamente melhor em GPU, especialmente quando envolve muitas iterações e estruturas simples de acesso à memória. No hardware utilizado, o custo por iteração da GPU não compensou o fluxo mais ágil da CPU.
- Assim, o estudo evidencia que a escolha do modelo de paralelização deve considerar o hardware disponível e o perfil computacional do algoritmo. Neste caso, o paralelismo leve da CPU superou soluções mais complexas de GPU, reforçando que mais threads nem sempre significam mais desempenho.

Obrigado

Repositório

<https://github.com/da-pet/kmeans-mpi>