# Paralelização do Algoritmo K-Means com OpenMP e CUDA

**Allan Guilherme G. Pego**[1], **Henrique P. Magalhães**[1], **Pedro M. Boscatti**[1]

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte/MG - Brasil

aggpego@sga.pucminas.br, henrique.magalhaes.1350435@sga.pucminas.br,

pedro.boscatti@sga.pucminas.br

## 1. Introdução

Este trabalho teve como objetivo implementar e paralelizar o algoritmo K-Means em C/C++, utilizando as tecnologias OpenMP e CUDA. O K-Means é um algoritmo de aprendizado não supervisionado para agrupamento de dados, com ampla aplicação em inteligência artificial.

O foco da atividade foi explorar diferentes formas de paralelismo em arquiteturas modernas de hardware:

- Execução paralela em CPU com OpenMP.
- Simulação de paralelismo para GPU usando OpenMP com atomics.
- Execução nativa em GPU utilizando CUDA.

## 2. Algoritmo K-Means

O algoritmo K-Means realiza o agrupamento de dados em k grupos baseando-se na proximidade de cada ponto ao centróide mais próximo. O funcionamento básico ocorre em etapas iterativas:

1. Inicialização dos grupos de forma aleatória.
2. Cálculo dos centróides de cada grupo.
3. Reatribuição dos pontos ao grupo com centróide mais próximo.
4. Repete as etapas 2 e 3 até convergência.

## 3. Ambiente de Testes

- **Hardware**: Intel i7 / GPU NVIDIA
- **SO**: Windows 10
- **Compiladores**:
  - gcc com `-fopenmp`
  - nvcc (CUDA Toolkit 11.8 ou superior)

## 4. Versões Implementadas

### 4.1. Versão Sequencial

Utiliza um código aberto como base, com implementação clássica do K-Means em C puro.
Link original: `https://github.com/Lakhan-Nad/KMeans-C`

### 4.2. OpenMP para CPU

Foram aplicadas diretivas `#pragma omp parallel for` e `reduction` para paralelizar:

- Inicialização dos clusters.
- Acúmulo das somas via buffers locais por thread.
- Reatribuição de grupos e contagem de alterações.

### 4.3. OpenMP (simulação de GPU)

- Utiliza `#pragma omp atomic` para somas concorrentes nos clusters.
- Executa em CPU, mas simula comportamento de paralelismo semelhante ao CUDA.

### 4.4. CUDA

Implementa 3 kernels:

- `assignPoints`: atribui grupo a cada ponto.
- `updateClusters`: soma os valores para centróides com `atomicAdd`.
- `averageClusters`: calcula os novos centróides.

## 5. Dados e Geração

- Pontos gerados aleatoriamente em distribuição polar.
- Total: 100.000 observações.
- Clusters: $k = 5$

## 6. Resultados Obtidos

### 6.1. Tabela de Tempo de Execução

| Versão | Tempo (s) |
|---|---|
| Sequencial | 22,929 |
| OpenMP CPU | 16,611 |
| OpenMP GPU* | 128,180 |
| CUDA | 36,240 |

*Simulação via atomics em CPU

### 6.2. Speedup (em relação à versão sequencial)

| Versão | Speedup |
|---|---|
| OpenMP CPU | 1,38x |
| OpenMP GPU | 0,18x |
| CUDA | 0,63x |

## 7. Análise e Discussão

- A versão OpenMP CPU teve desempenho melhor que a versão sequencial, mostrando ganho razoável com múltiplas threads.
- A versão com atomics (OpenMP-GPU) apresentou desempenho significativamente pior, devido à alta contenção nos `atomics` em CPU.
- A versão CUDA apresentou tempo melhor que a versão com atomics, mas ainda inferior à CPU multithreaded, indicando que o desempenho pode estar limitado por acesso à memória, overhead de kernel ou uso subótimo de blocos e threads.

## 8. Conclusão

O trabalho permitiu compreender as vantagens e desafios da paralelização de algoritmos de IA:

- OpenMP pode oferecer bons ganhos quando bem balanceado, especialmente em arquiteturas multicore.
- CUDA tem grande potencial, mas exige ajustes finos para superar desempenho de CPU em problemas menores.
- Nem toda paralelização garante aceleração — o balanceamento, overhead e arquitetura alvo impactam muito.

## 9. Referências


- Lakhan Nad: `https://github.com/Lakhan-Nad/KMeans-C`
- Documentação OpenMP: `https://www.openmp.org/`
- Documentação CUDA Toolkit: `https://docs.nvidia.com/cuda/`