Investigando speedup do algoritmo kmeans com OpenMP

Ítalo Epifânio

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

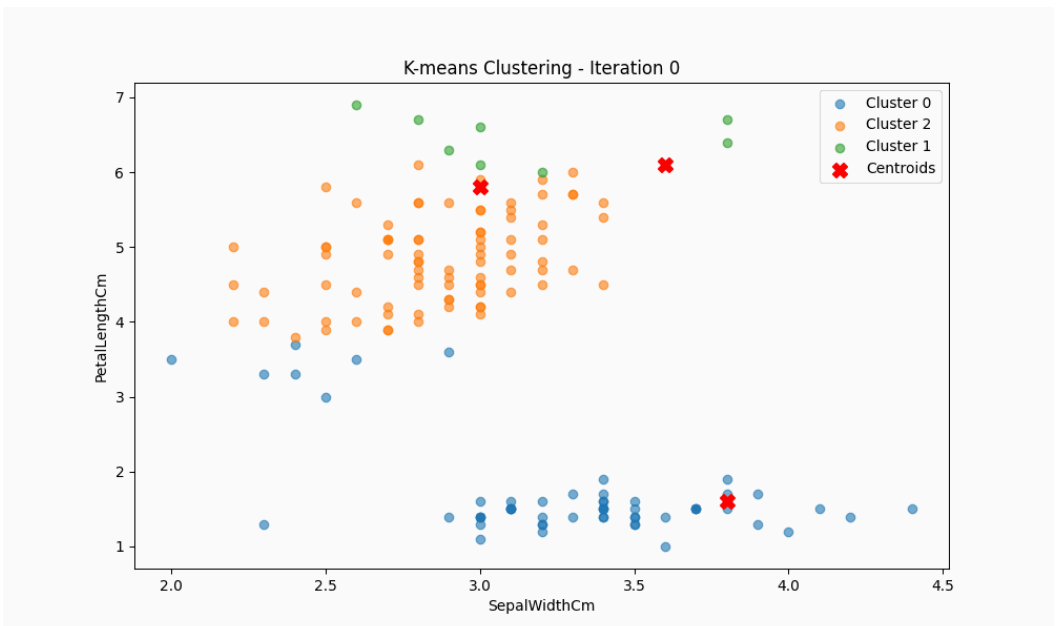
Introdução

- K-Means é um dos algoritmos mais usados em mineração de dados e aprendizado de máquina;
- Sua complexidade: $O(N \times D \times K \times I)$ pode tornar a sua execução em grandes datasets inviável;
- Objetivo: Avaliar os ganhos de paralelização em CPU com OpenMP.

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

K-Means Algorithm

1. Inicialize K centróides aleatóriamente
2. Atribua a cada ponto do dataset o centróide mais próximo (distância euclidiana)
3. Atualize os centróides baseado na média dos pontos atribuídos
4. Repita até a convergência



https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

Metodologia

- Estudo experimental
- Implementações em C com GCC 15.1 e OpenMP 5.1
- Testado em:
  - Intel i7-1165G7 (4 cores, 8 threads)
  - Ubuntu 22.04 LTS
- 30 execuções de cada experimento para significância estatística
- Python para análise de dados e execução

Dataset	N	D	K	N × D × K
Rice	3.809	7	2	53.326
HTRU2	17.898	8	2	286.368
WESAD	4.558.554	8	3	82.053.972

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

Referencial teórico

Trabalho	Algoritmos Implementados	Resultados
Daoudi et al. (2020)	Sequencial; CPU (OpenMP); GPU (OpenCL)	Na GPU speedup de 3,48 com K = 1024, D = 128 e N = 65.536; na CPU (OpenMP) speedup de 2 com K = 16 e D = 2.
Alghamdi et al. (2023)	Sequencial; Sequencial melhorado; CPU (OpenMP)	Na versão sequencial speedup entre 2,726 a 7,806; na sequencial melhorada speedup entre 3,613 a 9,014.
Praxedes et al. (2024)	Sequencial; GPU (CUDA)	Na GPU speedup entre 9,19 e 42,40 nos datasets utilizados no presente trabalho.

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

Algoritmo sequencial

```
Entrada: K (numeros de clusters), conjunto de N pontos com D dimensoes (dataset)
Saida: particoes dos N pontos em K clusters

// passo 1: inicializacao
inicialize k centroides randomicamente

dim = D
pontos = N

// passo 2: repetir ate convergir
enquanto nao convergiu:

    // passo 2a: fase de atribuicao
    para ponto em pontos:
        para centroide em centroides:
            centroideMaisProximo = menorDistanciaEuclidiana(ponto, centroide, dim)

        ponto.centroidMaisProximo = centroideMaisProximo

    // passo 2b: passo de atualizacao
    somaCentroides = [[]]
    contaCentroide = []
    para ponto em pontos:
        contaCentroide[ponto.centroidMaisProximo] += 1
        para d em dim:
            somaCentroides[ponto.centroidMaisProximo][d] += ponto
    para k em centroides:
        se contaCentroide[k] == 0:
            continue

        para d em dim: https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project
            centroides[k][d] = somaCentroides[k][d] / contaCentroide[k]
```

```
// passo 3: verificacao de convergencia
se diferenca(centroides, centroidesAnteriores) <= threshold:
    convergiu = verdadeiro
```

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

Algoritmo paralelo com OpenMP

```
Entrada: K (numeros de clusters), conjunto de N pontos com D dimensoes (dataset)
Saida: particoes dos N pontos em K clusters

// passo 1: inicializacao
inicialize k centroides randomicamente

dim = D
pontos = N

// passo 2: repetir ate convergir
enquanto nao convergiu:

    // passo 2a: fase de atribuicao
    #pragma omp parallel for schedule(static)
    para ponto em pontos:
        para centroide em centroides:
            #pragma omp simd reduction(+:sum)
            centroideMaisProximo = menorDistanciaEuclidiana(ponto, centroide, dim)

        ponto.centroidMaisProximo = centroideMaisProximo

    // passo 2b: passo de atualizacao
    somaCentroide = [[]]
    contaCentroide = []

    somaThread = [[[]]]
    contaThread = [[]]

    #pragma omp parallel for schedule(static)
    para ponto em pontos:
        tid = omp_get_thread_num()
        contaThread[tid][ponto.centroidMaisProximo] += 1
```

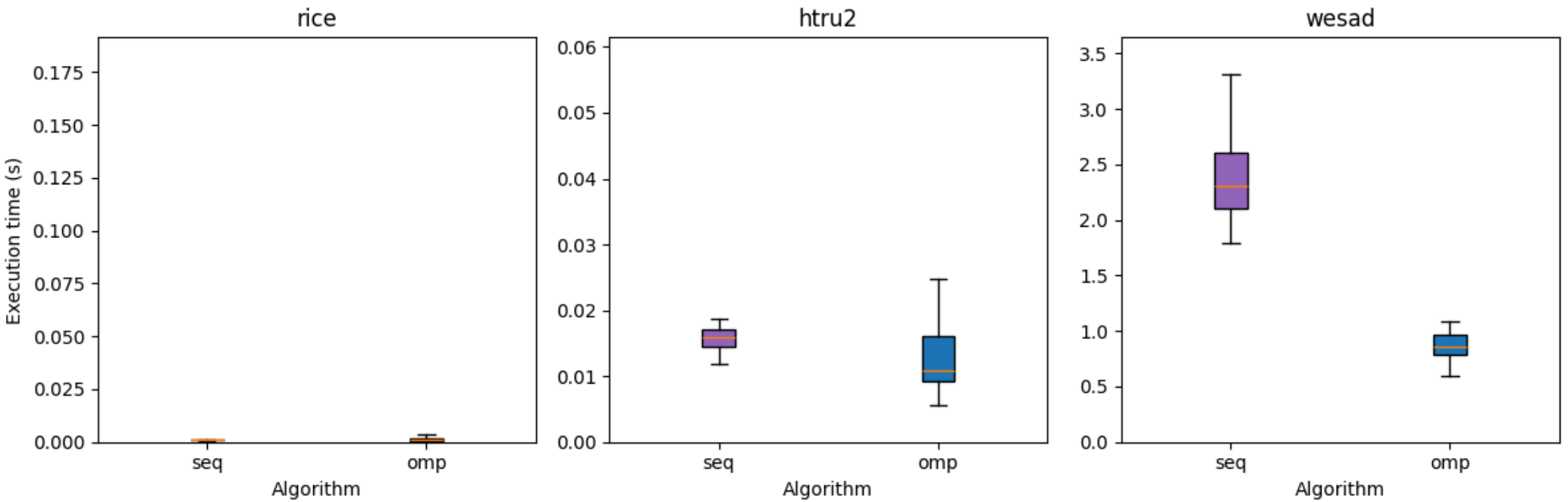
https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

```
    para d em dim:
        somaThread[tid][ponto.centroidMaisProximo][d] += ponto

// operacao reducao
para thread em threads:
    para centroide em centroides:
        contaCentroide[thread][centroide] += contaThread[thread][centroide]
        para d em dim:
            somaCentroide[centroide][d] += somaThread[thread][centroide][d]

para k em centroides:
    se contaCentroide[k] == 0:
        continue

    para d em dim:
        centroides[k][d] = somaCentroides[k][d] / contaCentroide[k]

// passo 3: verificacao de convergencia
se diferenca(centroides, centroidesAnteriores) <= threshold:
    convergiu = verdadeiro
```

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

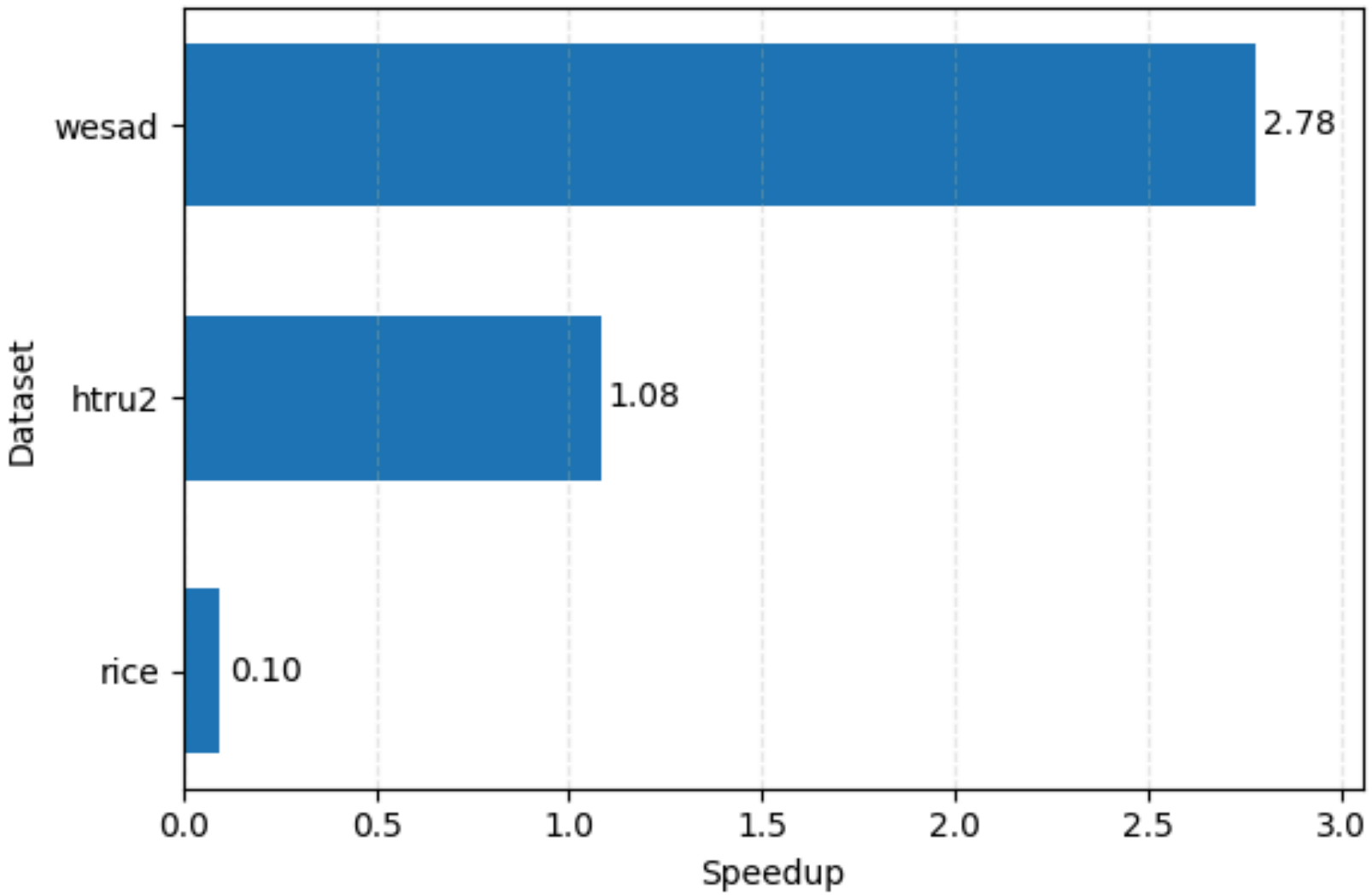
Resultados

Tempo de execução



https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

Speedup



https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project

Conclusão

- OpenMP traz ganhos relevantes de performance em datasets grandes;
- O estudo trouxe a hipótese de ganhos de SIMD para datasets com alta dimensão;
- Durante o desenvolvimento teve-se um problema de rápida convergência do algoritmo paralelo, devido ao fato da operação de redução se diferenciar do algoritmo sequencial, gerando um erro númerico mais significativo, “resolveu-se” através da conversão de “float” para “double”.

Trabalhos futuros

- Avaliar desempenho em datasets com alta dimensionalidade (D elevado);
- Aprofundar análise com métricas como eficiência paralela e balanceamento de carga;
- Realizar estudos de escalabilidade variando número de threads, K, N e D.

https://github.com/itepifanio/openmp-pthreads-cuda-study/tree/main/kmeans-project