



Paralelização do Algoritmo K-means em GPUs NVIDIA Utilizando a Biblioteca Numba

Trabalho de Conclusão de Curso

Vinícius Henrique Almeida Praxedes

Professor Orientador: Daniel Duarte Abdala

Professora Convidada 1: Júlia Tannús de Souza

Professor Convidado 2: Anderson Rodrigues dos Santos

Motivação

- Curiosidade de como utilizar **GPUs** para solucionar problemas matemáticos
- **GPGPU** → computação de propósito geral em GPUs
- Resolução de problemas lineares → Álgebra Linear
- Exemplo: multiplicação de matrizes

Motivação — Área Escolhida

- Outras áreas que lidam com problemas lineares:
 - a) **Big data**
 - b) Treinamento de redes neurais
- Foi escolhida a área de **big data**

Motivação — Agrupamento de Dados

- Data Mining

- Comumente usada no pré-processamento
- Classificação

Scalar

Vector

Matrix

Tensor

1

$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$

$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

$\begin{bmatrix} 1 & 2 & 3 & 2 \\ 1 & 7 & 5 & 4 \end{bmatrix}$

- Custo computacional alto para grandes datasets

- Muitas operações são vetoriais e independentes entre si → altamente paralelizáveis

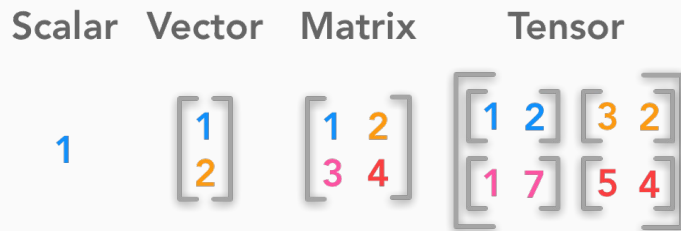
Motivação — Desenvolvimento para GPUs

- Processadores vetoriais comuns e acessíveis

- Inicialmente, GPGPU era difícil

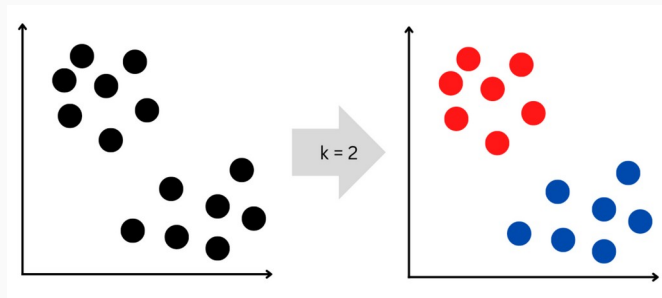
- DirectX
- OpenGL

- Facilitando: plataforma **CUDA** da NVIDIA (C, **C++**, Fortran) → Biblioteca **Numba (Python)**



Motivação — K-Means

- Algoritmo de agrupamento antigo (1957)
- Ainda amplamente utilizado
- Conceito simples
- Já implementado e paralelizado diversas vezes



Paralelização do Algoritmo K-means

em GPUs NVIDIA

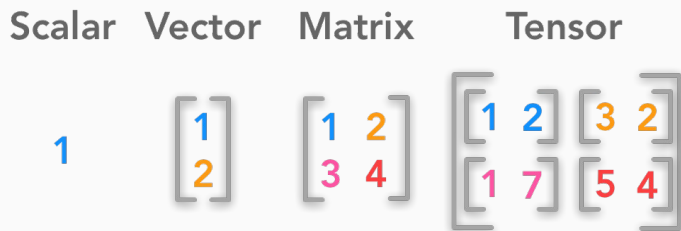
Utilizando a Biblioteca Numba

Sumário

- CPUs vs GPUs
- Paralelização
- Agrupamento de dados: k-means
- Plataforma CUDA
- Biblioteca Numba
- Implementação: k-means serial vs. k-means paralelo
- Experimentos: datasets e resultados
- Conclusão

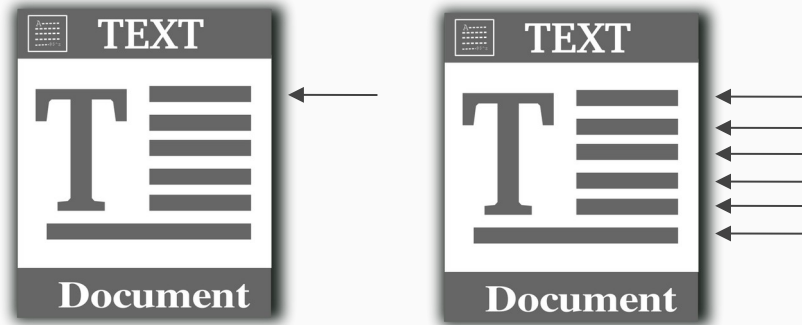
CPUs vs GPUs

- CPUs → dezenas de núcleos otimizados para operações escalares
- GPUs → milhares a dezenas de milhares de núcleos otimizados para operações vetoriais



Problemas paralelizáveis

- Pesquisa de um termo num texto



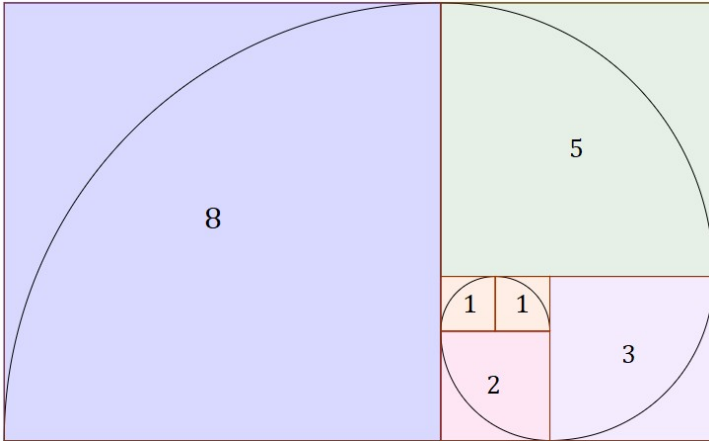
- Multiplicação de matrizes

The diagram shows the multiplication of matrix $[A]$ (2x3) and matrix $[B]$ (3x2). Matrix $[A]$ has rows $[a_{11} \ a_{12} \ a_{13}]$ and $[a_{21} \ a_{22} \ a_{23}]$. Matrix $[B]$ has columns $[b_{11} \ b_{21} \ b_{31}]$ and $[b_{12} \ b_{22} \ b_{32}]$. The resulting matrix $[A] \times [B]$ has elements L_1C_1 and L_1C_2 in the first row, and L_2C_1 and L_2C_2 in the second row. The calculation for L_1C_1 is shown below:

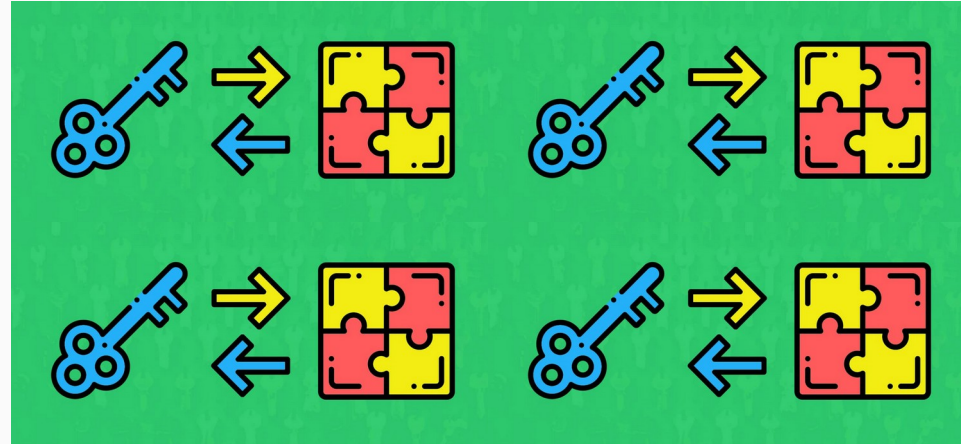
$$L_1C_1 = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Problemas não paralelizáveis

- Encontrar n-ésimo fibonacci



- Hashing consecutivo

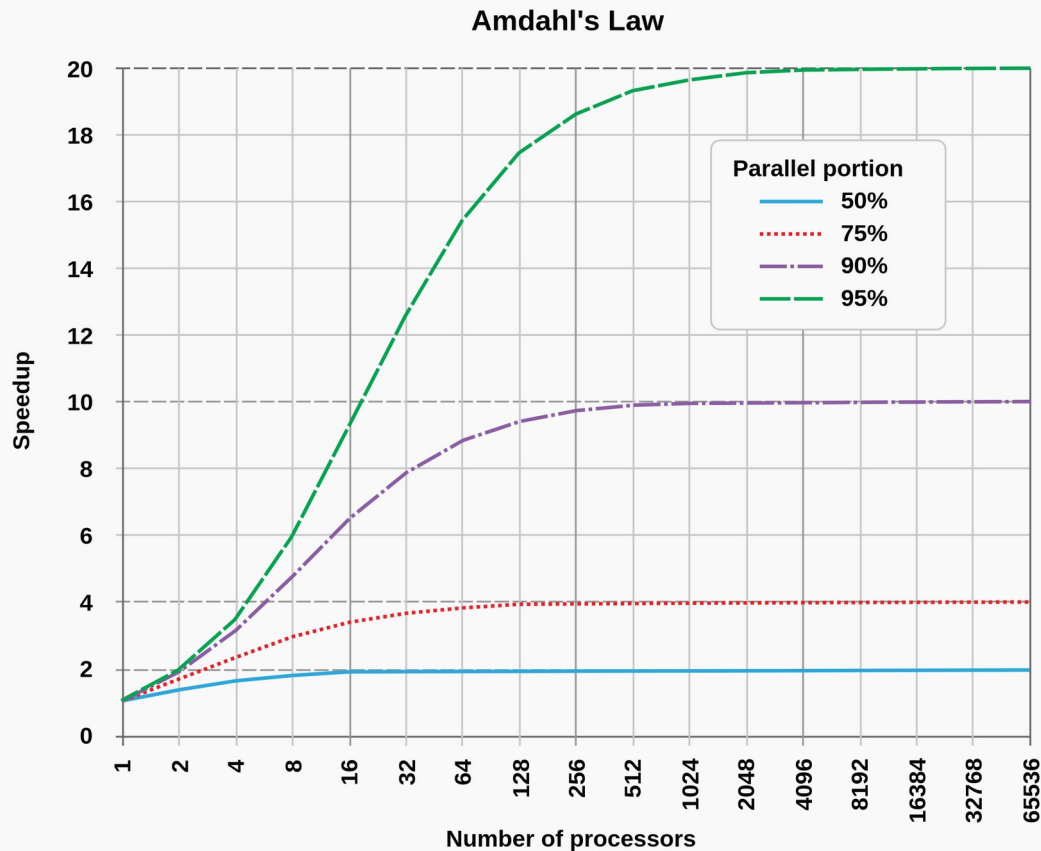


Limite Teórico — Lei de Amdahl

- Descreve aumento de velocidade máximo de um algoritmo através da paralelização

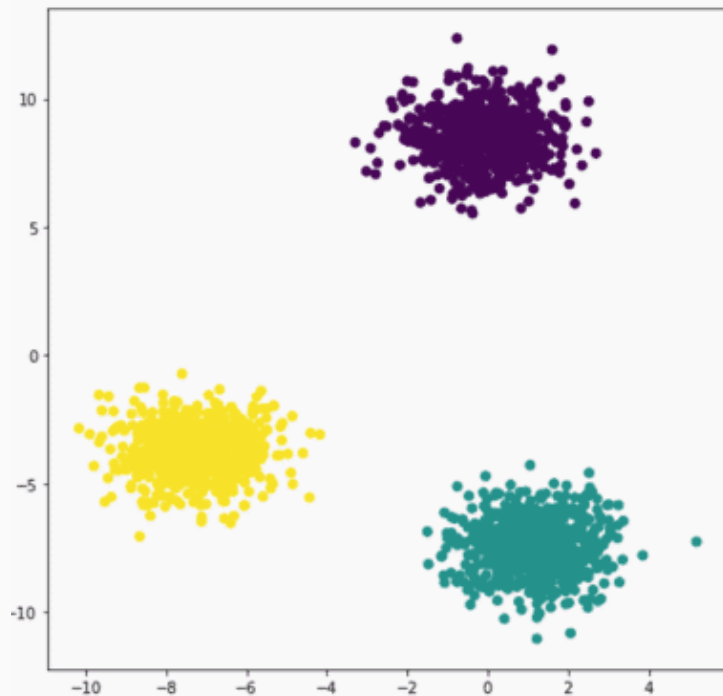
$$\frac{1}{1 - \frac{T_p}{T}}$$

- T_p = tempo gasto em operações paralelizáveis
- T = tempo total gasto na execução

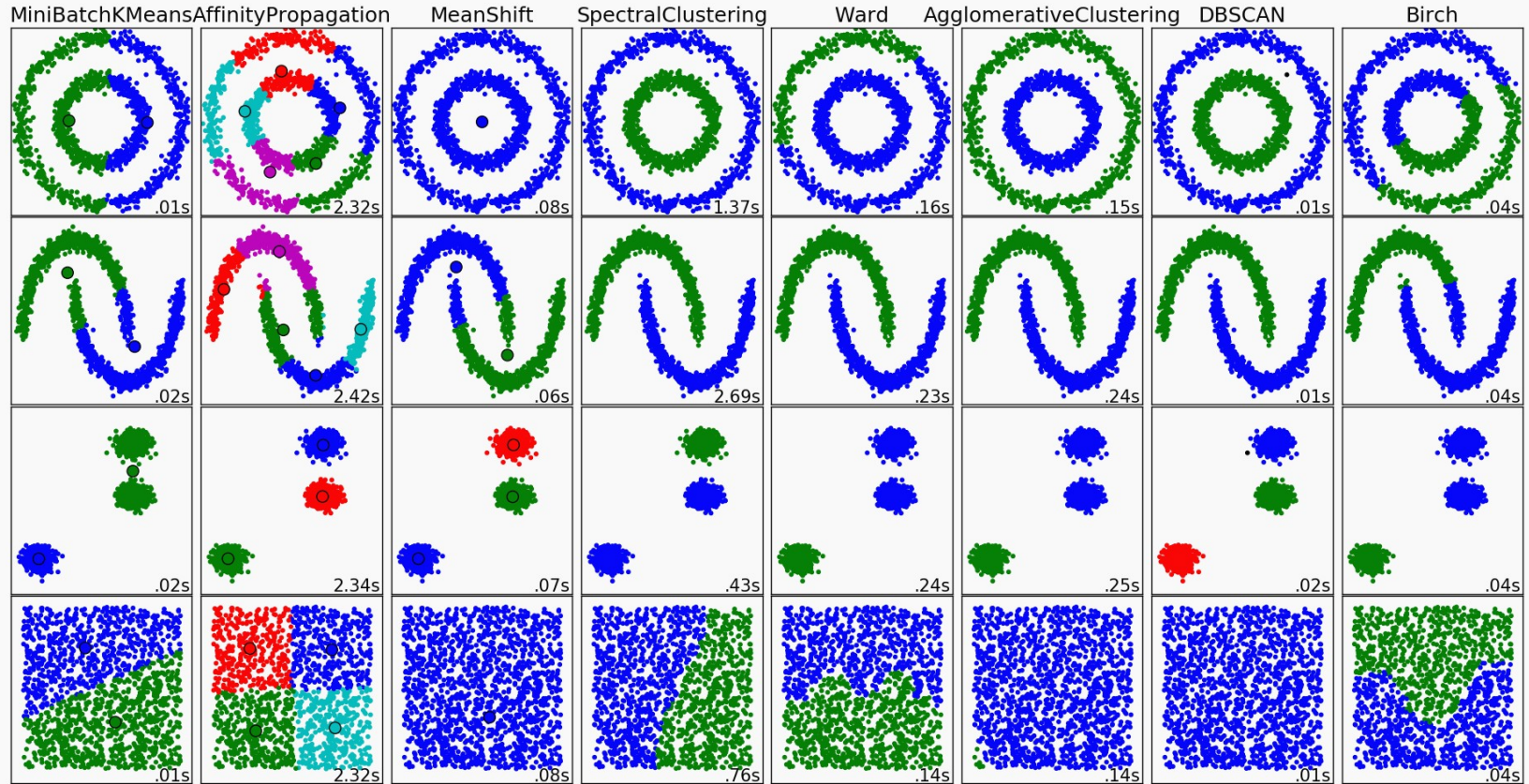


Algoritmos de Agrupamento

- Agrupar elementos de um conjunto de dados
 - Maximizar diferença inter-grupo
 - Minimizar diferença intra-grupo
- Diferença e semelhança → conceitos abstratos
- Utilizado em diversas áreas
 - Data mining
 - Classificação
 - Aprendizado de máquina
 - Processamento de imagem
 - Geração de recomendações



Algoritmos de Agrupamento



K-means

Entrada

$P = \{P_1, P_2, \dots, P_n\}$, um conjunto de N objetos de dados (pontos em um espaço D -dimensional);

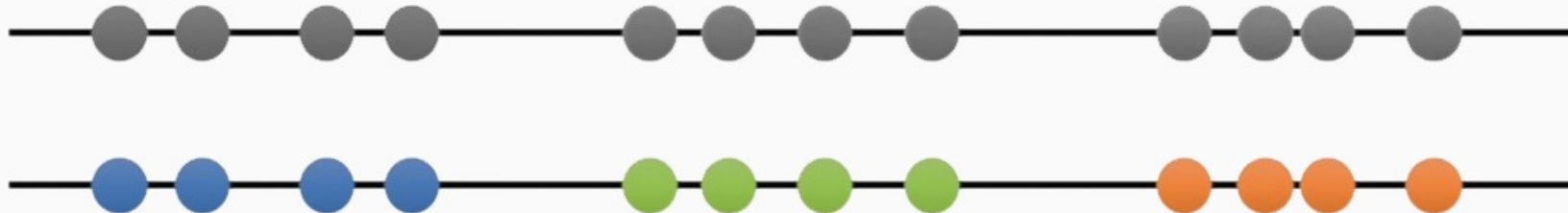
K , o número de agrupamentos desejado;

I_{max} , o número de iterações máximas do algoritmo

Saída

Um conjunto de K agrupamentos, onde cada um dos N objetos em P está associado a exatamente um conjunto.

$N = 12; K = 3; D = 1$



K-means

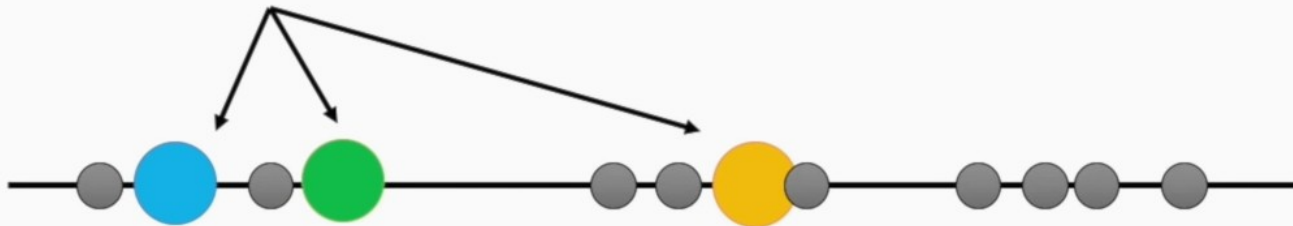
Passos

- 1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
- 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

Não há mudança entre os centroides da iteração atual e o da anterior;
OU...

O número de iterações realizadas ultrapassa um máximo I_{max} .



K-means

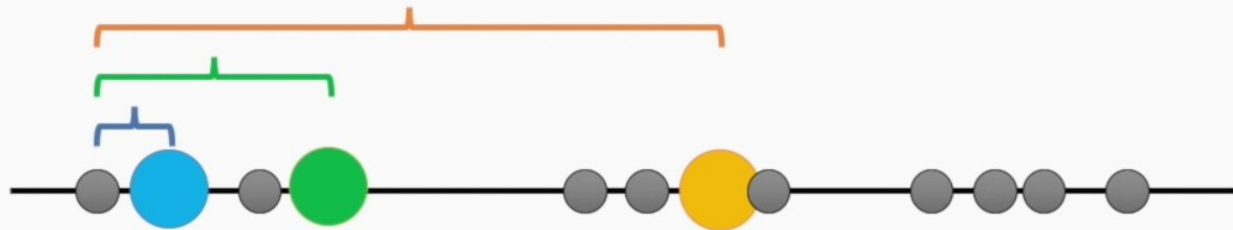
Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

Não há mudança entre os centroides da iteração atual e o da anterior;
OU...

O número de iterações realizadas ultrapassa um máximo I_{max} .



K-means

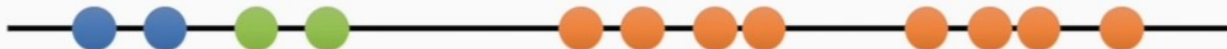
Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Crítérios de convergência

Não há mudança entre os centroides da iteração atual e o da anterior;
OU...

O número de iterações realizadas ultrapassa um máximo I_{max} .



K-means

Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

Não há mudança entre os centroides da iteração atual e o da anterior;
OU...

O número de iterações realizadas ultrapassa um máximo I_{max} .



K-means

Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

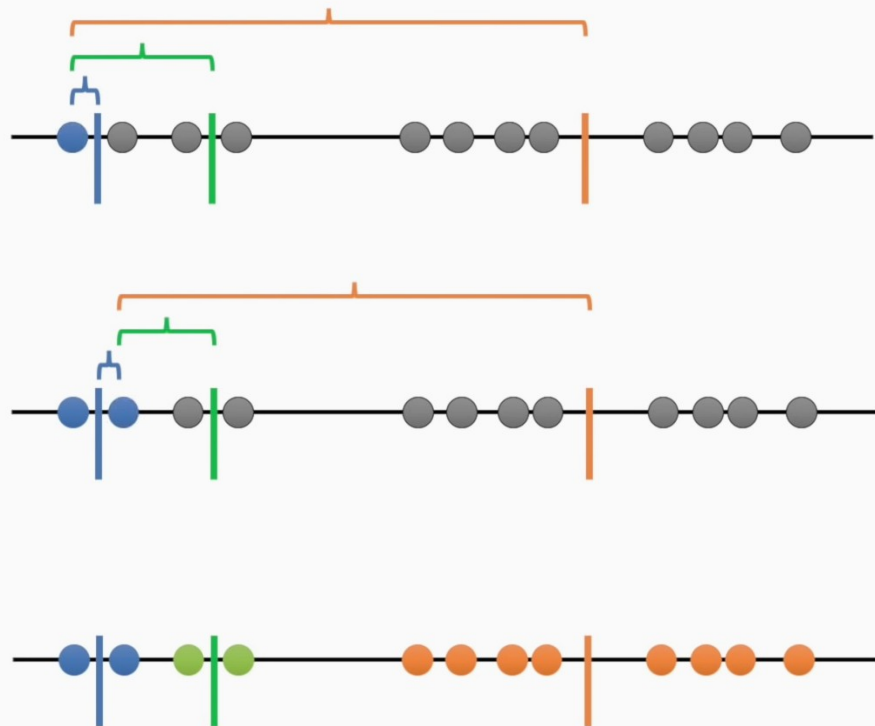
Não há mudança entre os centroides da iteração atual e o da anterior;
OU...

O número de iterações realizadas ultrapassa um máximo I_{max} .



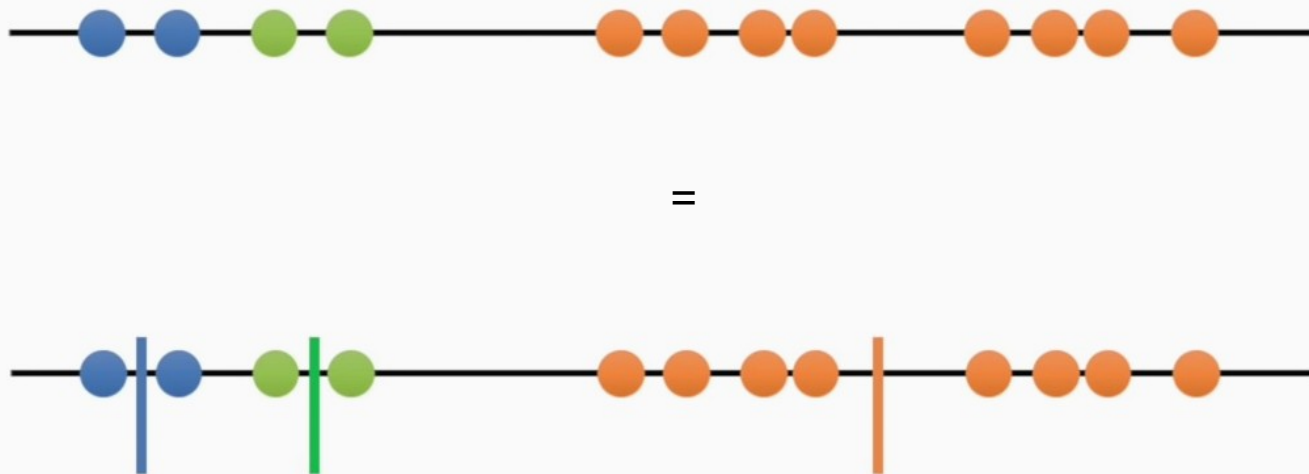
K-means

- Repetindo...

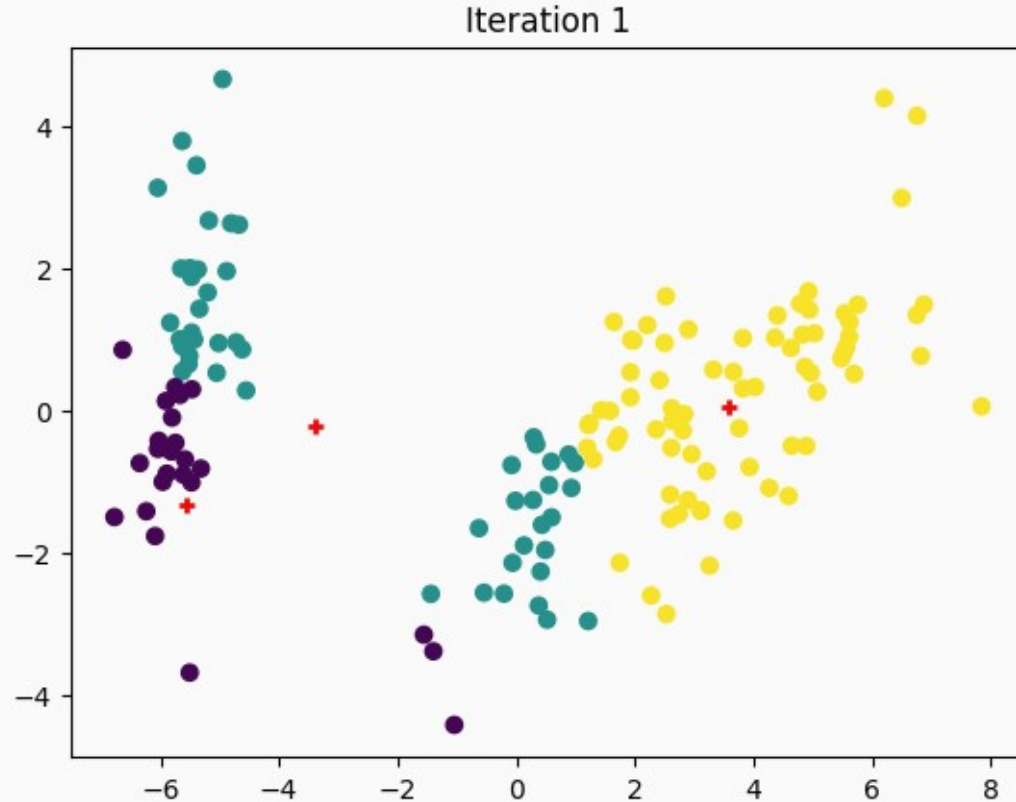


K-means

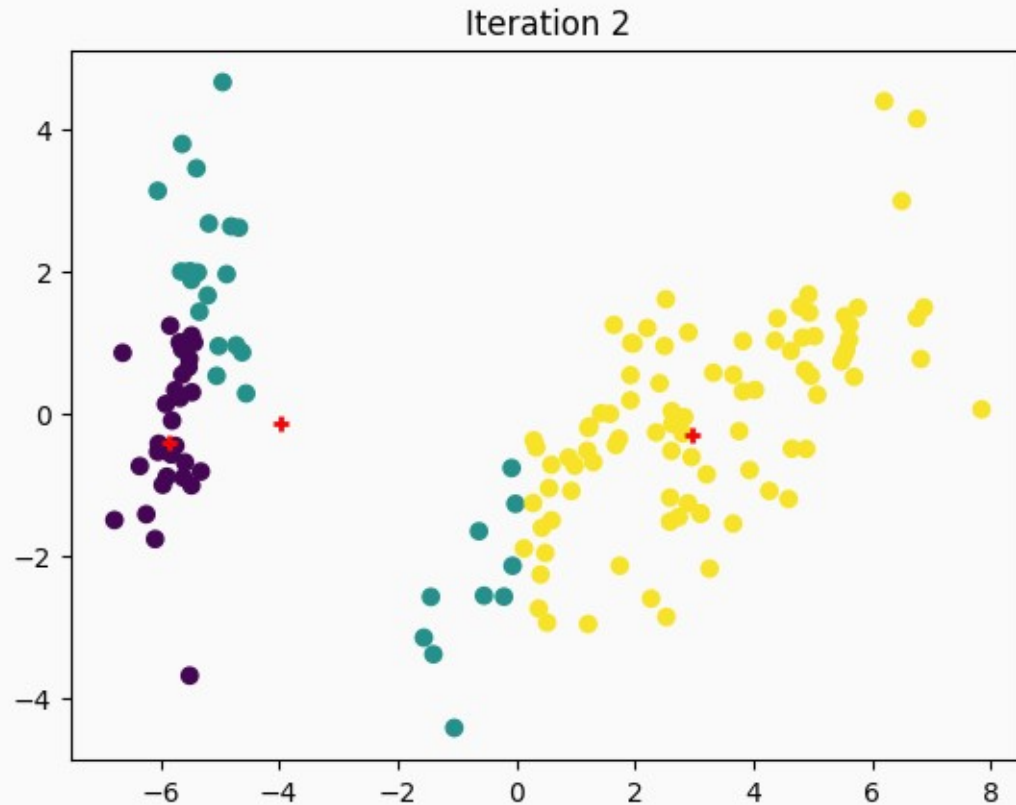
- Parada quando não há mudança nos centroides ou foi atingido o limite de iterações



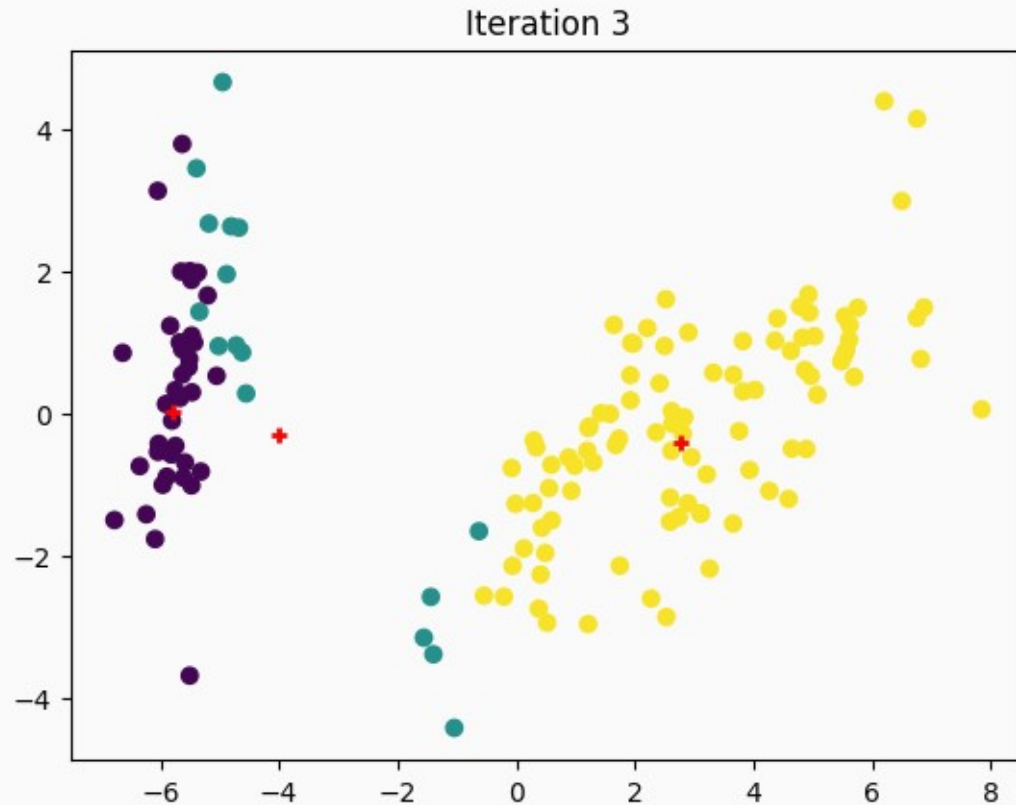
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



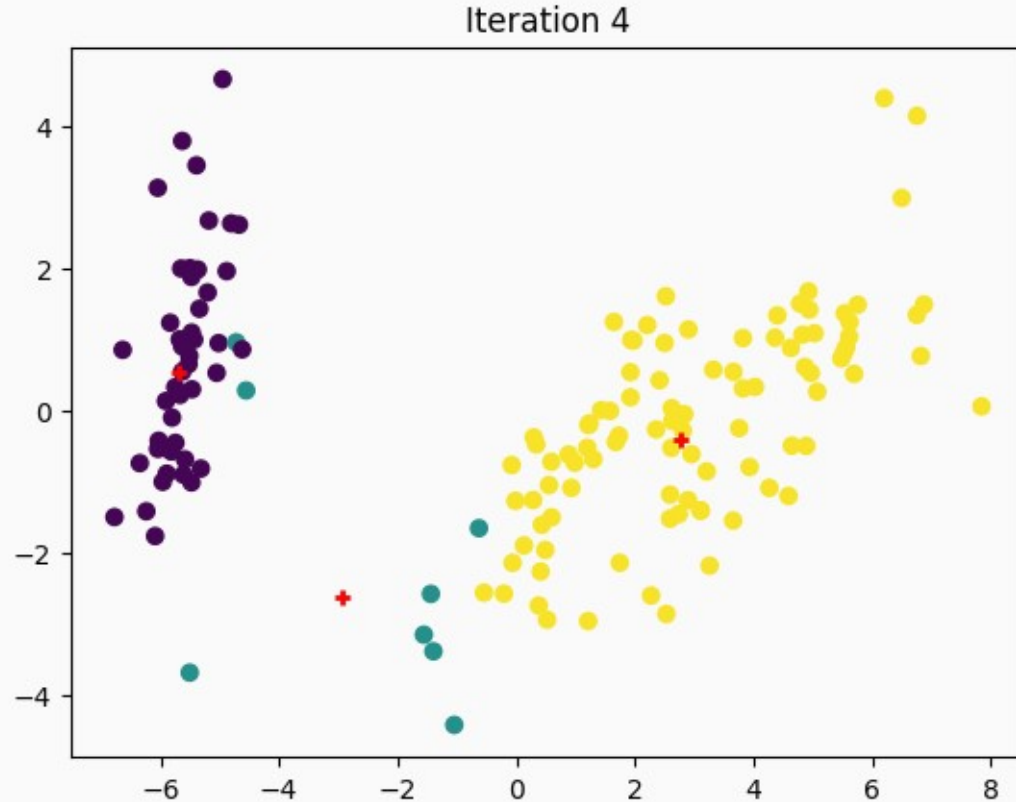
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



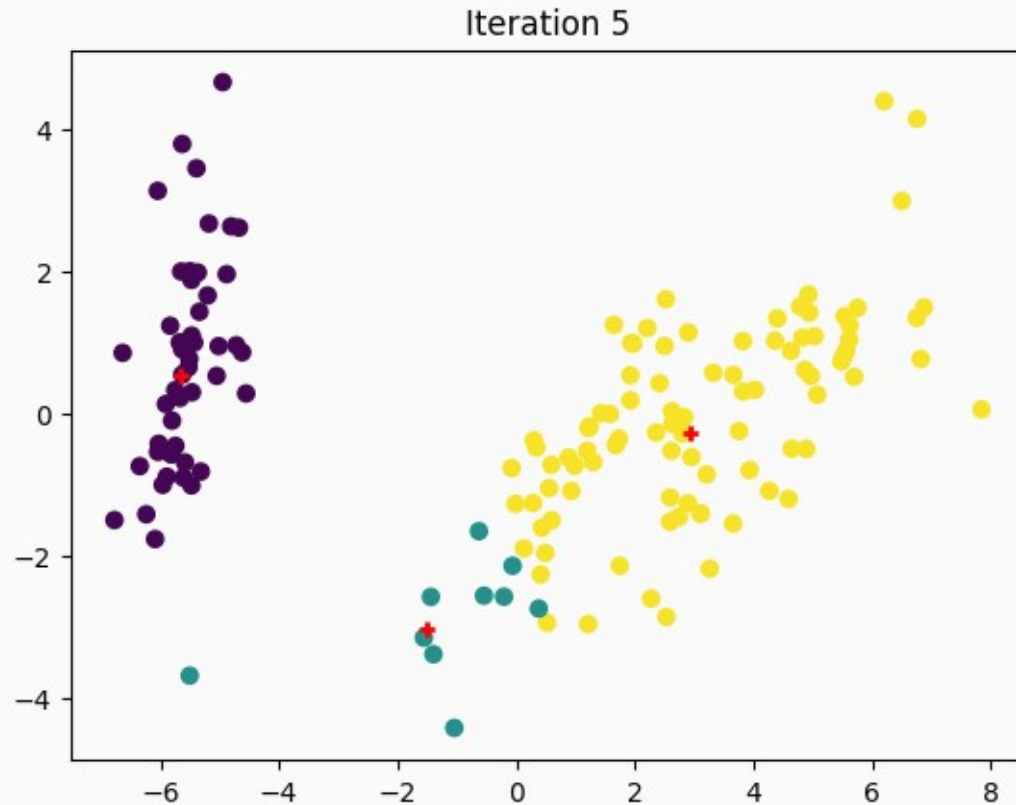
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



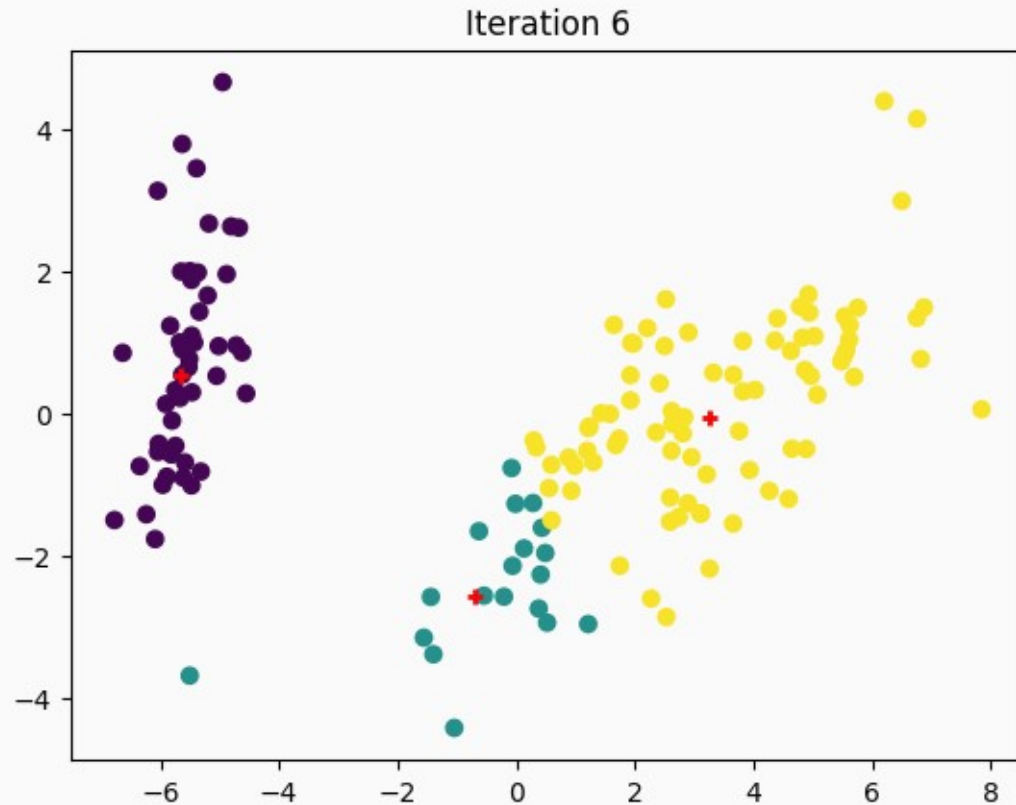
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



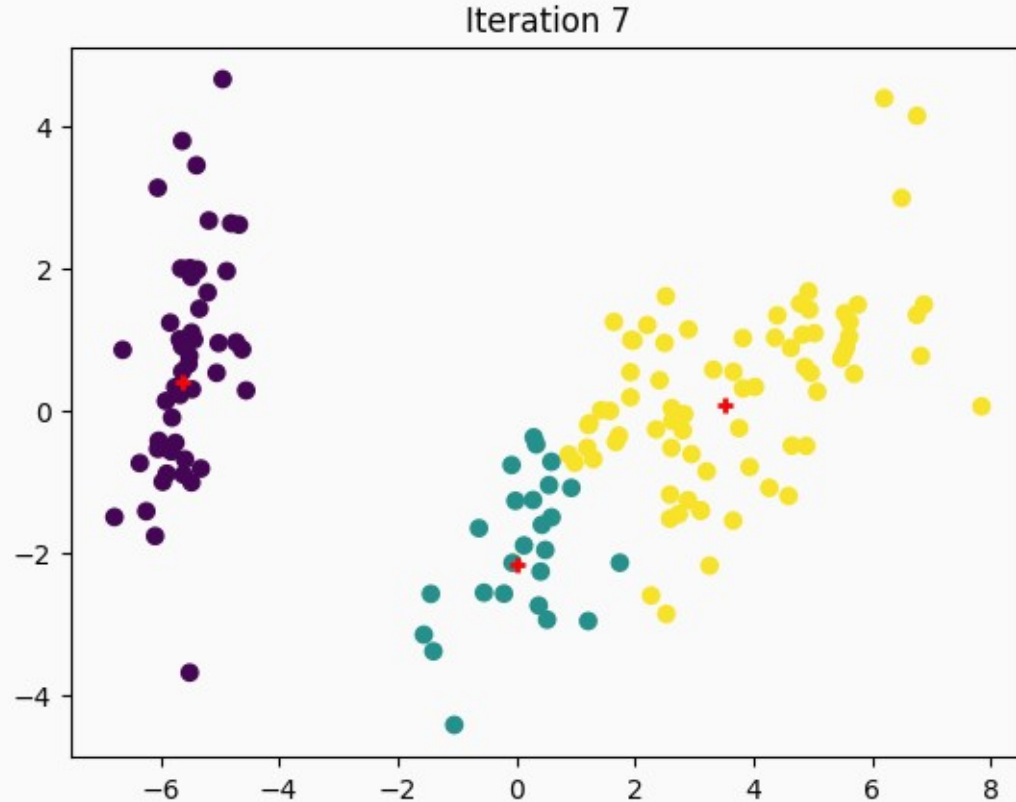
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



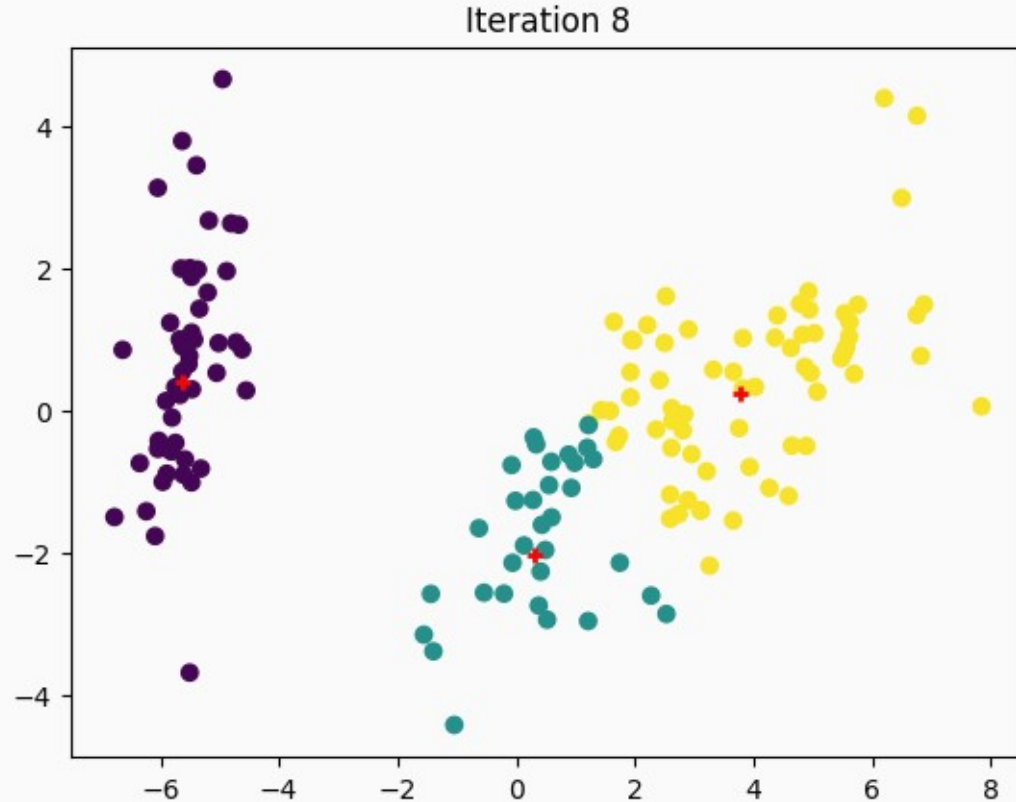
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



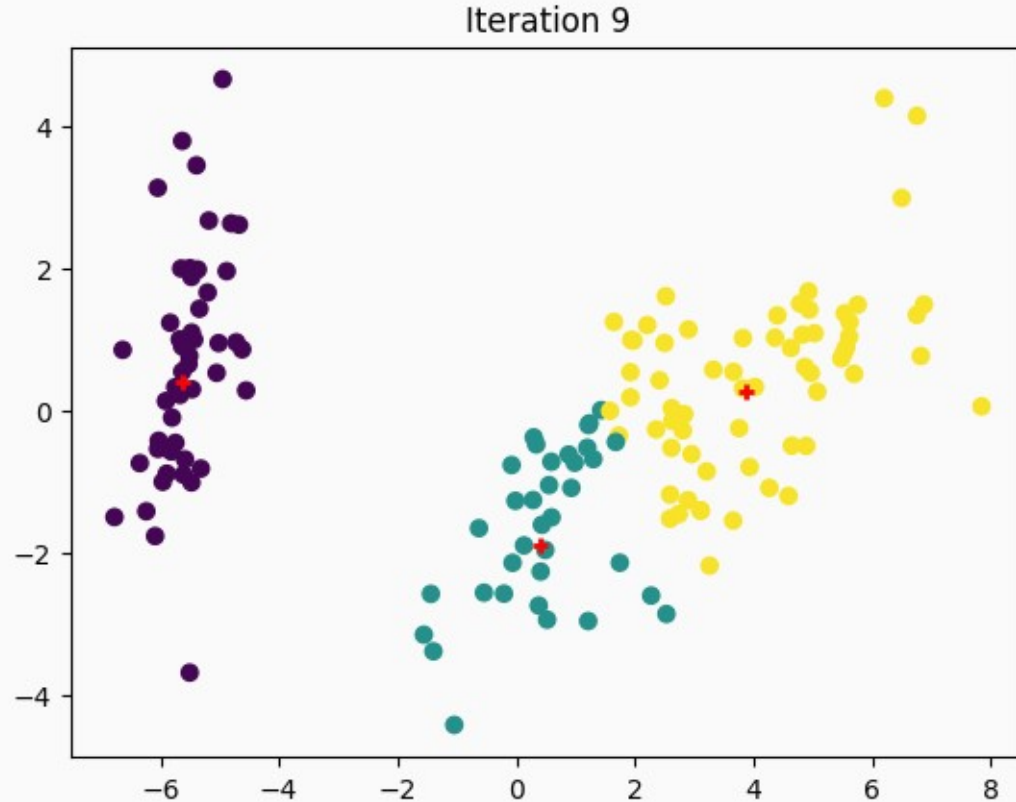
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



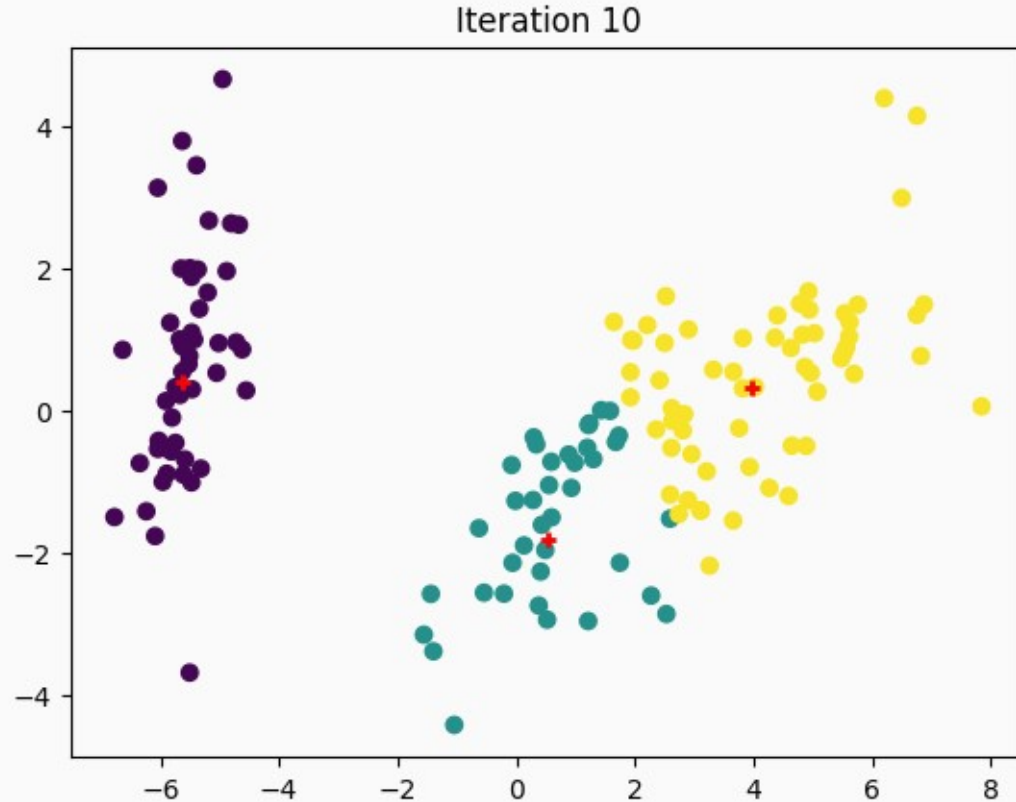
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



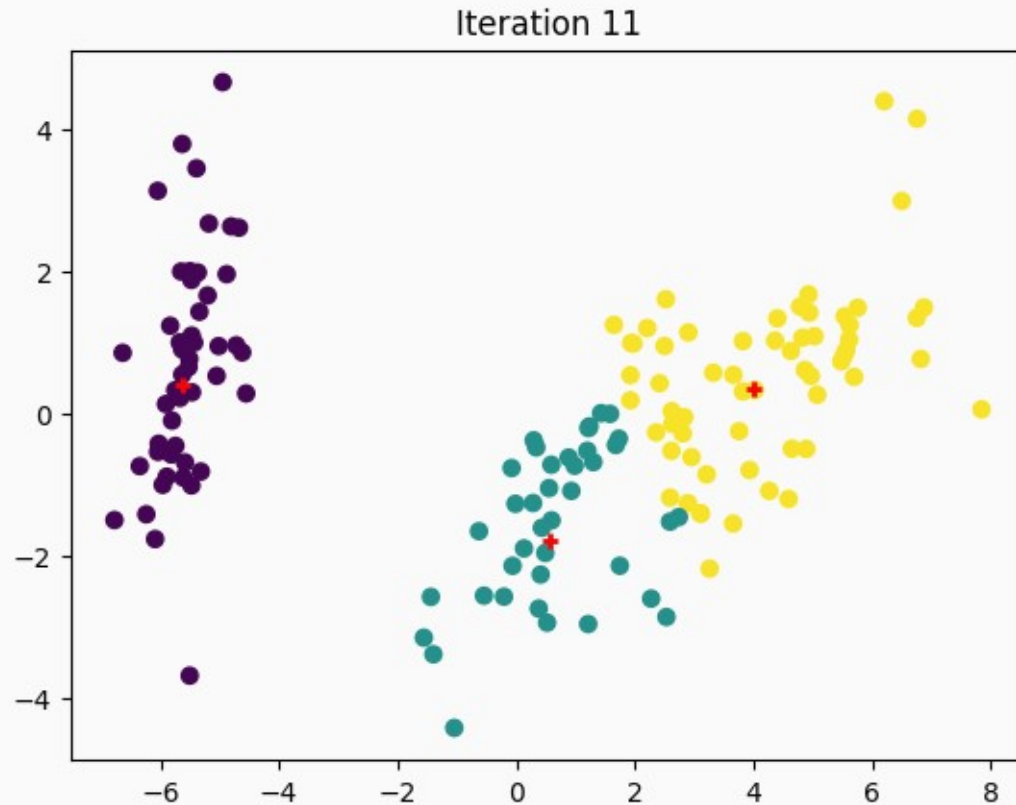
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



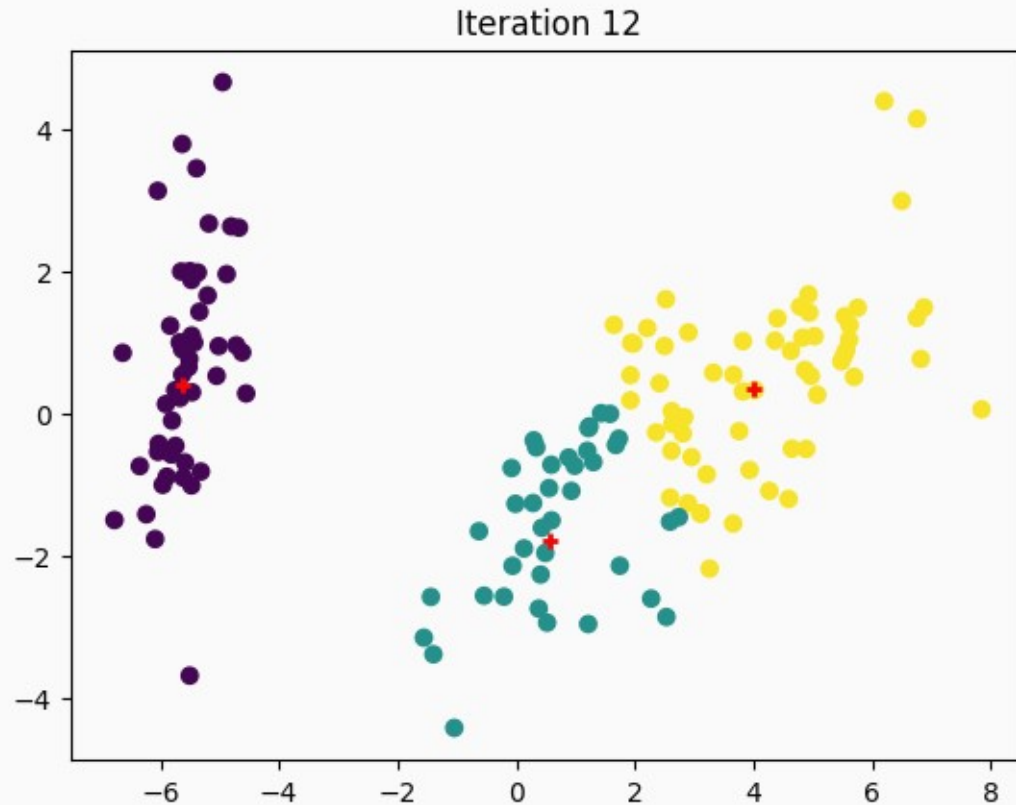
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$



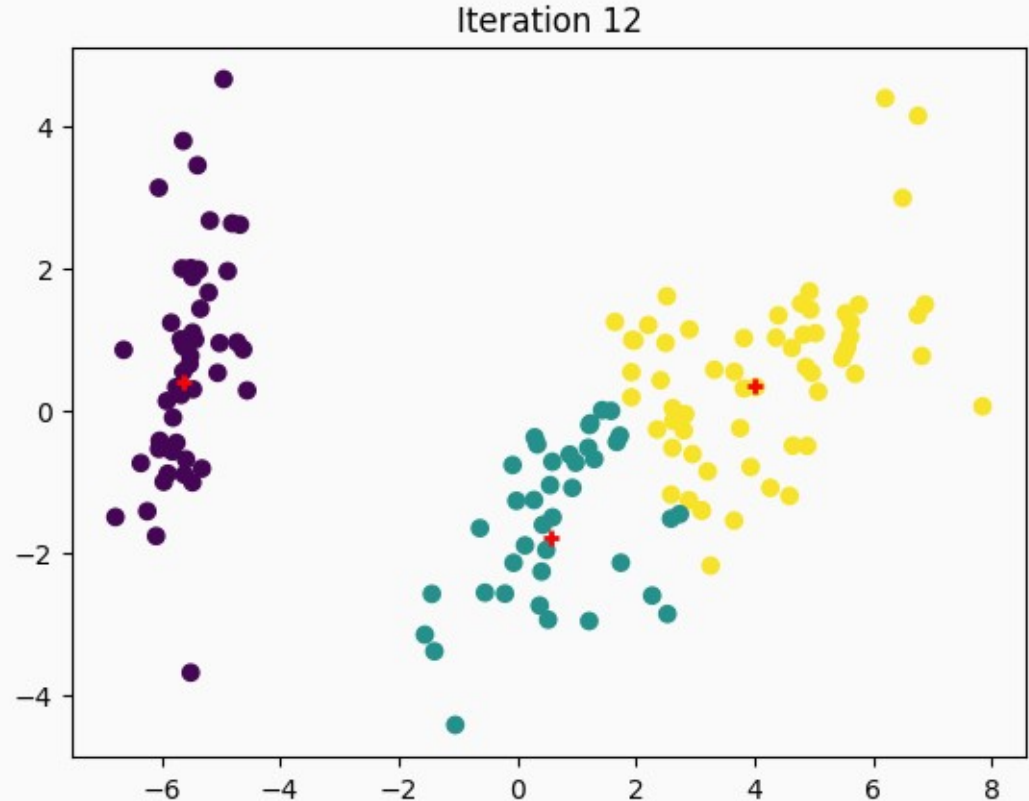
K-means — Dataset real (Iris) — $N = 150$; $D = 4$; $K = 3$

Acertos:
133/150, 88,66%

Iris-setosa:
50/50, 100%

Iris-versicolor:
36/50, 72%

Iris-virginica:
47/50, 94%



K-means — Partes Paralelizáveis

Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

- Não há mudança entre os centroides da iteração atual e o da anterior;
- OU...
- O número de iterações realizadas ultrapassa um máximo I_{max} .

K-means — Partes Paralelizáveis

- Cálculo das distâncias $\rightarrow O(N * K * I)$

Passos

1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;

2. Repita (até que os critérios de convergência sejam atingidos):

→ 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;

2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência


Não há mudança entre os centroides da iteração atual e o da anterior;
OU...

O número de iterações realizadas ultrapassa um máximo I_{max} .

K-means — Partes Paralelizáveis

- Cálculo das médias $\rightarrow O(N * D * I)$

Passos


1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 -  2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

- Não há mudança entre os centroides da iteração atual e o da anterior;
- OU...
- O número de iterações realizadas ultrapassa um máximo I_{max} .

- Seleção dos centroides iniciais? $\rightarrow O(K) \rightarrow K \ll N$, ganho irrisório

Passos

- 
1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
 2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.


Critérios de convergência

- Não há mudança entre os centroides da iteração atual e o da anterior;
- OU...
- O número de iterações realizadas ultrapassa um máximo I_{max} .

K-means — Partes Não-paralelizáveis

- Iterações separadas do laço de repetição $\rightarrow O(I)$

Passos


- 
1. Escolha arbitrariamente K pontos em P para servirem de centroides iniciais dos agrupamentos;
 2. Repita (até que os critérios de convergência sejam atingidos):
 - 2.1 Atribua cada ponto de P ao agrupamento que possui o centroide mais próximo, calculado pela distância euclidiana;
 - 2.2 Calcule novos centroides para cada agrupamento através da média das coordenadas de todos os pontos do grupo.

Critérios de convergência

- Não há mudança entre os centroides da iteração atual e o da anterior;
- OU...
- O número de iterações realizadas ultrapassa um máximo I_{max} .

NVIDIA CUDA

- Compute Unified Device Architecture (Arquitetura de Dispositivo de Computação Unificada)
- API para C, **C++** e Fortran
- Permite utilização do paralelismo e processamento vetorial de GPUs NVIDIA

CUDA C

Standard C Code	Parallel C Code
<pre>void saxpy_serial(int n, float a, float *x, float *y) { for (int i = 0; i < n; ++i) y[i] = a*x[i] + y[i]; } // Perform SAXPY on 1M elements saxpy_serial(4096*256, 2.0, x, y);</pre>	<pre><u>__global__</u> void saxpy_parallel(int n, float a, float *x, float *y) { int i = blockIdx.x*blockDim.x + threadIdx.x; if (i < n) y[i] = a*x[i] + y[i]; } // Perform SAXPY on 1M elements saxpy_parallel<<<4096, 256>>>>(n, 2.0, x, y);</pre>

<http://developer.nvidia.com/cuda-toolkit>

Implementação Serial — Soma de Vetores (C++)

```
1  #include <iostream>
2  #include <math.h>
3
4  // Função que adiciona os elementos de dois vetores
5  void add(int n, float *x, float *y){
6      for (int i = 0; i < n; i++)
7          y[i] += x[i];
8  }
9
10 int main(void){
11     int N = 1<<28; // 268.435.456 elementos
12
13     float *x = new float[N];
14     float *y = new float[N];
15
16     // Inicializar vetores no host
17     for (int i = 0; i < N; i++) {
18         x[i] = 3.77f; y[i] = 3.23f;
19     }
20
21     // Rodar na CPU
22     add(N, x, y);
23
24     // Checar se há erros (todos os valores devem ser 7.0)
25     float maxError = 0.0f;
26     for (int i = 0; i < N; i++)
27         maxError = fmax(maxError, fabs(y[i] - 7.0f));
28     std::cout << "Max error: " << maxError << "\n";
29
30     // Liberar memória
31     delete [] x;
32     delete [] y;
33
34     return 0;
35 }
```

Implementação Paralela — Soma de Vetores (C++ e CUDA)

```
1  #include <iostream>
2  #include <math.h>
3
4  __global__
5  void add(int n, float *x, float *y){
6      int index = blockIdx.x * blockDim.x + threadIdx.x;
7      int stride = blockDim.x * gridDim.x;
8      for (int i = index; i < n; i += stride)
9          y[i] += x[i];
10 }
11
12 int main(void){
13     int N = 1<<28; // 268.435.456 elementos
14
15     float *x, *y;
16     cudaMallocManaged(&x, N*sizeof(float));
17     cudaMallocManaged(&y, N*sizeof(float));
18
```

```
19     for (int i = 0; i < N; i++) {
20         x[i] = 3.77f; y[i] = 3.23f;
21     }
22
23     int blockSize = 1024;
24     int numBlocks = ceil(N / blockSize);
25
26     add<<<numBlocks, blockSize>>>(N, x, y);
27     cudaDeviceSynchronize();
28
29     float maxError = 0.0f;
30     for (int i = 0; i < N; i++)
31         maxError = fmax(maxError, fabs(y[i] - 7.0f));
32     std::cout << "Max error: " << maxError << "\n";
33
34     cudaFree(x);
35     cudaFree(y);
36
37     return 0;
38 }
```

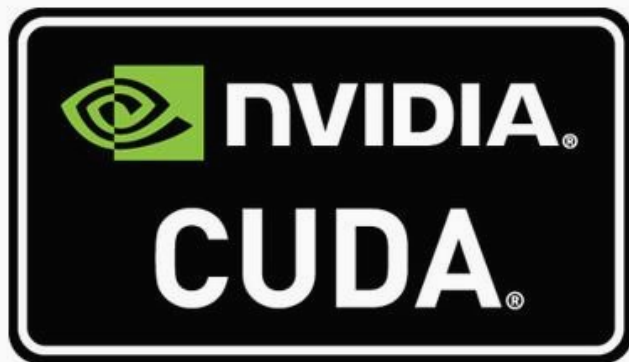
- blockIdx.x = ID do bloco na GPU
- threadIdx.x = ID da thread no bloco
- gridDim.x = quantidade de blocos na GPU
- blockDim.x = quantidade de threads por bloco

CUDA — Speed-up da Soma de Vetores

- Tempo médio em 100 execuções
- Versão serial C++: **308 ms**
- Versão paralela CUDA: **11 ms**
- Speed-up: **28x**
- Erro máximo: **0.0**



- Manipulação direta de memória da GPU
- Aritmética complexa para que cada thread se “localize”
- Limitado às linguagens C, C++ e Fortran



Biblioteca Numba

- Compilador e biblioteca para **Python**
- Tradução de código **Python** para código de máquina otimizado
- Velocidades se aproximam de **C** e **Fortran**
- Suporte ao CUDA para rodar algoritmos em GPU



Implementação Serial — Soma de Vetores 2D (Python e Numpy)

```
1  import time; import numpy as np
2
3  def addArrayCPU(a, b):
4      return a + b
5
6  def checkMaxErr(c):
7      # Checando erro máximo (todos elementos devem ser 42.0):
8      minRow = c.min(axis=0)
9      maxRow = c.max(axis=0)
10     maxErr = 0.0
11     for dIdx in range(D):
12         maxErr = max(maxErr, abs(42.0 - minRow[dIdx]))
13         maxErr = max(maxErr, abs(42.0 - maxRow[dIdx]))
14     print(f'Max error: {maxErr}')
15
16     D = 2**2
17     N = int(2**28 * 1.5) // D # N * D = 402.653.184 elementos
18
19     # Inicializando vetores
20     a = np.full((N, D), 27.2, np.float32)
21     b = np.full((N, D), 14.8, np.float32)
22
23     # Realizando adição
24     c = addArrayCPU(a, b)
25
26     checkMaxErr(c)
27
```

Implementação Paralela — Soma de Vetores 2D (Python, Numpy e Numba)

```
1  import time; import numpy as np; import numba
2
3  @numba.guvectorize(
4      ['void(float32[:],float32[:],float32[:])'],
5      '(d),(d)→(d)', nopython=True, target='cuda'
6  )
7  def addArrayGPU(a, b, c):
8      d = len(a)
9      for dIdx in range(d):
10         c[dIdx] = a[dIdx] + b[dIdx]
11
12 > def checkMaxErr(c): ...
13
14
15
16
17
18
19
20
21
22 D = 2**2
23 N = int(2**28 * 1.5) // D # N * D = 268.435.456 elementos
24
25 # Inicializando vetores
26 a = np.full((N, D), 27.2, np.float32)
27 b = np.full((N, D), 14.8, np.float32)
28
29 # Inicializando vetor de retorno
30 c = np.zeros((N, D), np.float32)
31
32 # Realizando adição
33 addArrayGPU(a, b, c)
34
35 checkMaxErr(c)
36
```


Numba — Speed-up da Soma de Vetores

- Tempo médio em 100 execuções
- Versão serial Python: **2.501,24 ms**
- Versão paralela Numba: **397,50 ms**
- Speed-up: **6,29x**
- Erro máximo: **0.0**



Paralelizando o K-means

- Cálculo de distâncias — versão serial em Python (Numpy e Pandas)

```
11 | while iteration ≤ maxIter and not centroids_OLD.equals(centroids):  
12 |     distances = centroids.apply(lambda x: np.sqrt(((dataset - x)  
    |     ** 2).sum(axis=1))))
```

- Aplica-se à todos os centroides uma função lambda que calcula a distância euclidiana de todos os pontos do dataset para este centroide
- dataset \rightarrow (N, D); centroids \rightarrow (K, D); .sum() \rightarrow (N); distances = (N, K)

Paralelizando o K-means

- Cálculo de distâncias — versão paralela em Python (Numpy e Numba)

```
63     while iteration ≤ maxIter and not np.array_equal  
        (centroids_OLD__np ,centroids__np):  
64         distances = np.zeros((n, k))  
65         calcDistances(centroids__np, dataset__np, distances)
```

- centroids__np → (K, D); dataset__np → (N, D); distances = (N, K)

Paralelizando o K-means

- Função calcDistances()

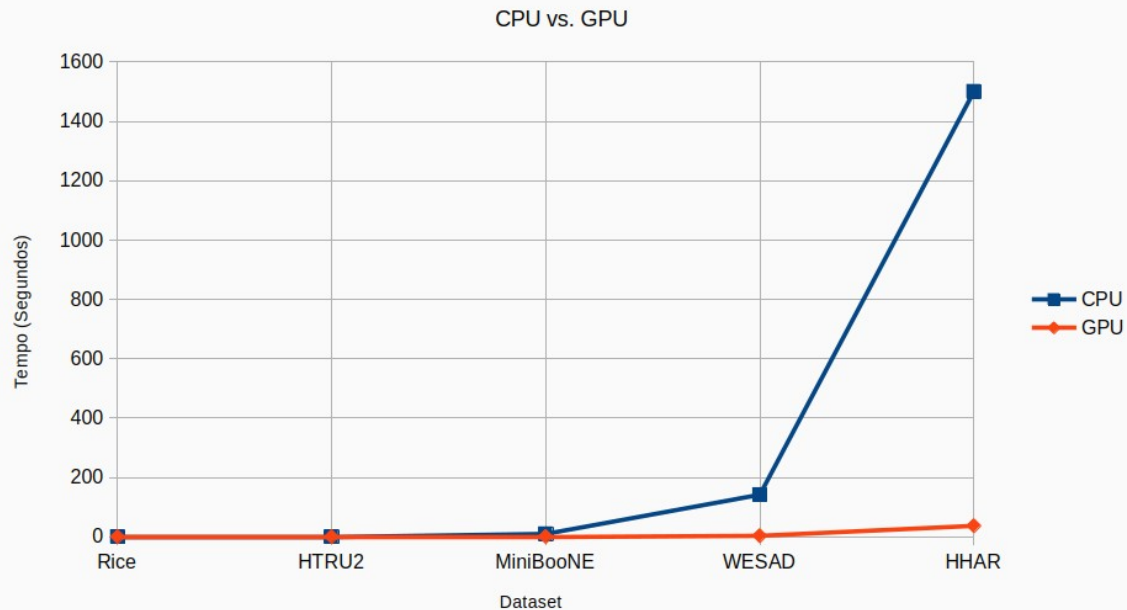
```
11 @numba.guvectorize(  
12     ['void(float64[:, :], float64[:, :], float64[:, :])'],  
13     '(k,d),(d)→(k)', nopython=True, target='cuda'  
14 )  
15 def calcDistances(centroids, rowDataset, rowResults):  
16     d = len(rowDataset)  
17     for centroidIndex, centroid in enumerate(centroids):  
18         distance = 0.0  
19         for dim in range(d): distance += (rowDataset[dim] - centroid[dim]) ** 2  
20         distance = distance ** (1/2)  
21         rowResults[centroidIndex] = distance
```

Experimentos — Datasets

- Experimentos realizados em cinco datasets reais
- Repositório de Machine Learning da Universidade da Califórnia em Irving (UCI)

Dataset	N	D	K	$N \cdot D \cdot K$	Modificação?
Rice	3.810	7	2	53.340	Não
HTRU2	17.898	8	2	286.368	Não
MiniBooNE	129.596	50	2	12.959.600	Remoção de outliers
WESAD	4.588.552	8	3	110.125.248	Sub-conjunto
HHAR	13.932.632	3	7	292.585.272	Sub-conjunto

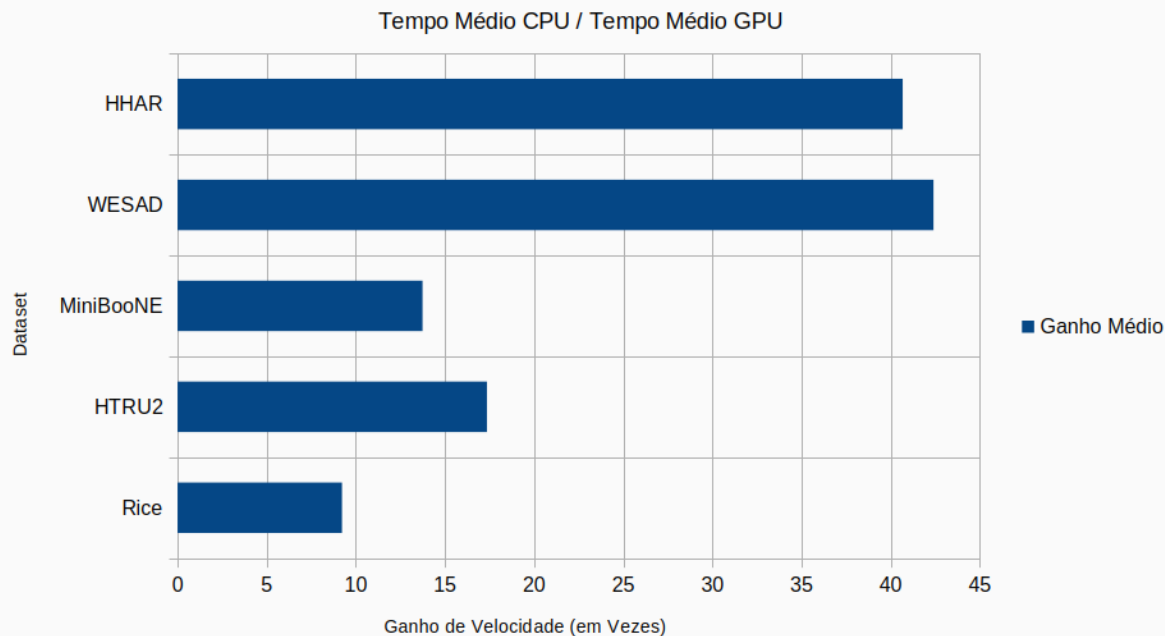
Experimentos — Speed-up



Experimentos — Speed-up

Dataset	Execuções	Tempo Médio CPU (S)	Tempo Médio GPU (S)
Rice	100	0.0800	0.0087
HTRU2	100	0.3327	0.0192
MiniBooNE	100	9.6021	0.6988
WESAD	20	141.7909	3.3438
HHAR	10	1500.6155	36.8901

Experimentos — Speed-up



Experimentos — Precisão

Dataset	Execuções	P. Média (CPU)	P. Média (GPU)	Melhora
Rice	100	91.0433%	91.3821%	+0.372180
HTRU2	100	91.7588%	91.7589%	+0.000061
MiniBooNE	50	50.8039%	50.8288%	+0.048967
WESAD	5	63.5576%	65.0151%	+2.293257
HHAR	2	28.3104%	28.3097%	-0.002485

Conclusões

- Ganhos de velocidade obtidos dentro da ordem de magnitude esperada de 3x à 80x
- Utilizando ferramentas de alto-nível de abstração: Python e Numba
- Acesso facilitado à área de GPGPU → grande ferramenta para *Data Science*
- Ganhos não afetam a precisão do algoritmo



Monografia, códigos e mais material
complementar disponíveis no GitHub:

<https://github.com/vinivosh/ufu-tcc2/>