Clusterização utilizando K-medoids em paralelo

Filipe Leuch Bonfim GRR20092368

flb09@inf.ufpr.br

UFPR - Universidade Federal do Paraná - Curitiba - PR - Brasil

Resumo. *Este relatório visa mostrar a implementação do Algoritmo de Clusterização K-medoids, ultilizando a arquitetura Cuda, da Nvidia, e efetuar comparações de desempenho entre a sua versão seqüencial e paralela.*

Abstract. *This report aims to show the implementation of clustering algorithm K-medoids using the CUDA architecture,from Nvidia, and make performance comparisons between their sequential and parallel version.*

1. Introdução

Algoritmos de aglomeração visam identificar grupos homogêneos de objetos com base nos seus dados, e geralmente efetuam operações comuns sobre estes objetos. Estes algoritmos são ultilizados em vários tipos de aplicações, tais como: mineração de dados, visão computacional e análise de dados, em outras palavras, "Algoritmos de Clusterização dividem os dados em grupos úteis ou significativos, chamados clusters, nos quais a similaridade intra- cluster é maximizada e a similaridade inter-cluster é minimizada. Estes clusters descobertos podem ser usados para explicar as características da distribuição dos dados subjacentes e assim servir como base para várias técnicas de análise e mineração de dados. As aplicações de clusterização incluem caracterização de diferentes grupos de clientes baseado nos padrões de compra, categorização de documentos na World Wide Web, agrupamento de genes e proteínas que possuem funcionalidades similares, agrupamento de localizações geográficas propensas a terremotos através de dados sismológicos, etc.", este trecho foi retirado do artigo de [BELTRAME 2010].

2. O Algoritmo de K-Medoids

K-Medoids é o algoritmo de agrupamento (clustering) que será ultilizado para segmentar imagens em k-clusters em paralelo, sendo k a quantidade de segmentos de tons de cinza, pois foram ultilizadas imagens no formato PGM.

Funcionamento do algoritmo:

1. São selecionados k pontos iniciais aleatórios da imagem, que servirão como medoids.
2. Cada ponto é rotulado com medoid mais próximo, e o custo (cost) é calculado.
3. Os medoids são recalculados de maneira aleatória entre os pontos que não eram medoids e é feita uma cópia dos antigos medoids.
4. Os novos medoids são utilizados para recalcular os custos.
5. Se o novo custo for maior que o anterior, então o algoritmo chega ao fim.
6. O passo 2 ao 5 são repetidos até que não haja mudanção, em relação ao custo dos medoids.

3. Estratégia de implementação

3.1. Algoritmo Serial

Primeiramente foi definido um vetor de medoids através de pontos existentes na imagem de forma aleatória, e sem repetição, e então foi efetuada uma primeira rotulação, que é feita através de uma matriz de rotução do mesmo tamanho da imagem original, nesta rotulação cada ponto, é associado ao medoid com menor diferença, e o medoid escolhido é atribuído na mesma posição deste ponto na matriz de rotulação, e no final foi obtido um custo inicial.

Fórmula para o calculo do Custo, aonde M é o conjunto de medoids, X o conjunto de dados, K a quantidade de clusters e N o total de dados:

$$Cost(M, X) = \sum_{i=1}^{n} \min_{j=1}^{k}(d(m_j, x_i))$$

A partir disto, o algoritmo inicia uma seqüencia de n iterações, sendo n um número previamente definido, para obtenção de um mellhor custo, ou seja, a cada iteração é definido um novo medoid, que é escolhido de forma aleatória e o custo é recalculado. Ao final, foi ultilizado o vetor de medoids que obteve o menor custo para a rotulação. Abaixo alguns trechos de código referente ao recálculo do custo:

```
1
2
3  /* Funcao que define medoid com a menor diferenca */
4  int escolhe_caso(int medoids[],int x,int tam, int *dist)
5  {
6          int i, ind, dist1, dist2;
7
8          ind = 0;
9
10         dist1 = fabs(medoids[0] - x);
11         for( i = 1; i < tam; i++)
12         {
13                 dist2 = fabs(medoids[i] - x);
14                 if(dist2 < dist1)
15                 {
16                         dist1 = dist2;
17                         ind = i;
18                 }
19         }
20         *dist = dist1;
21         return medoids[ind];
22 }
23
24
25 /*Funcao que percorre a imagem e obtem novo custo*/
26 int calcula_custo(img_struct nova_img, int tam,int medoids[])
27 {
```

```
28          int i, j, novo_custo, dist, k;
29
30          i = j = novo_custo = 0;
31
32          while(i<nova_img.altura)
33          {
34                  j=0;
35                  while(j<nova_img.largura)
36                  {
37                          /*escolhe a que grupo pertence a posicao atual*/
38                          k=escolhe_caso(medoids,nova_img.pontos[j+i*nova_img.largura],tam,&dist);
39                          novo_custo+=dist;
40                          j++;
41                  }
42                  i++;
43          }
44          return novo_custo;
45  }
```

3.2. Algoritmo Paralelo

A principal idéia em relação a pararelização deste algoritmo, foi a de atribuir a cada thread um ponto da imagem, e fazer com que ela calcule a menor distância deste ponto, em relação ao vetor de medoids.

Após cada thread calcular a menor distância, este resultado é colocado em um vetor, que tem o tamanho igual a quantidade de threads, sendo este vetor indexado pelo ID de cada thread, em relação ao grid. Após isto é feito um reduce da soma, de todos os resultados obtidos, utilizando uma função da biblioteca thrust pentecente ao cuda. Para que isto ocorra, primeiro o vetor de resultados,chamado d_vcustos, é alocado no device, e em seguida este é associado a um ponteiro do tipo dvice_ptr, para que o vetor possa ser utilizado na função reduce sum da biblioteca thrust. Abaixo podemos ver como isto foi feito:

```
1
2           thrust::device_ptr<int> cptr = thrust::device_pointer_cast(d_vcustos);
3           /* Funcao reduce sum da biblioteca thrust*/
4           novo_custo = thrust::reduce(cptr, cptr + nPontos);
```

Para que cada thread fique responsável por cada ponto da imagem, foi utilizada a mesma estratégia sugerida na parte do código referente a multiplicação de matrizes, em que é criada uma abstração de duas dimensões ultilizando a função dim3, e que está contida na documentação oficial. De inicio, foi definido um bloco de tamanho 16 por 16, e então a imagem é divida pela quantidade de blocos necessários para que cada ponto receba uma thread. Também foram utilizadas as funções cudaMalloc() e cudaMemcpy(), para alocar e copiar a imagem, o vetor de medoids, e o vetor de resultados na memória do device.

```
1           /*
```

```
2                   Aloca espaco para vetor de medoids no device,
3                   sendo tam igual ao numero de clusters
4           */
5           cudaMalloc(&d_medoids,tam*sizeof(int));
6
7           /*
8                   Aloca vetor com a quantidade de pontos da imagem,
9                   sendo nPontos o numero total de pontos
10          */
11          cudaMalloc(&d_vcustos,nPontos*sizeof(int));
12
13          /*
14                  Aloca os pontos da imagem
15          */
16          cudaMalloc(&d_nova_img.pontos, nPontos*sizeof(int));
17
18          /*
19                  Copia para a memria do device os valores dos pontos
20          */
21      cudaMemcpy(d_nova_img.pontos, nova_img.pontos, nPontos*sizeof(int),cudaMemcpyHostToDevice);
22
23      /* Definicao das dimensoes do Grid e do Block*/
24          dim3 dimBlock(16, 16);
25          dim3 dimGrid(nova_img.largura / dimBlock.x, nova_img.altura / dimBlock.y);
```

Na função execultada pelo device, primeiramente é obtido o ID referente ao grid da thread, e então é efetuado o calculo para se descobrir a menor distancia referente ao vetor de medoids, e então o resultado é colocado no indice do vetor de resultados(vcustos) correspondente a thread.

```
1
2
3   __global__ void calcula_custo(img_struct nova_img, int tam,int *medoids,int *vcustos){
4           int j,dist1,dist2,x,tid;
5
6           int lin = blockIdx.y * blockDim.y + threadIdx.y;
7       int col = blockIdx.x * blockDim.x + threadIdx.x;
8       tid = lin*nova_img.largura + col; /*ID global da thread*/
9           x = nova_img.pontos[tid];
10
11          dist1 = fabsf(medoids[0] - x);
12          for( j = 1; j < tam; j++)
13          {
14                  dist2 = fabsf(medoids[j] - x);
15                  if(dist2 < dist1)
16                          dist1 = dist2;
17          }
18          vcustos[tid] = dist1;
19          __syncthreads();
```

3.3. Resultados Dos Testes

Para a realização dos testes foram considerados 3 fatores: a quantidade de clusters(k), pois quanto maior for, aumentará o número do cálculo de distâncias em relação aos medoids por ponto; a quantidade de iterações(n), pois quanto maior for, mais vezes a imagem será percorrida; e por último, o tamanho da imagem, para verificar o comportamento do algoritmo para diferentes números de pontos.

3.4. Variando a Quantidade de Clusters

Para este teste foi definida foram definidos alguns padrões, que serão listados abaixo:

- Resolução da Imagem: 1920x 1200
- Iterações(N): 100

K(Clusters)	Execução Seqüencial	Execução Paralela	SpeedUP
10	0m13.147s	0m0.981s	13.402
20	0m24.550s	0m1.368s	17.950
50	0m56.134s	0m2.653s	21.159
100	1m44.593s	0m4.174s	25.058
200	3m18.411s	0m7.189s	27.599

Tabela 1: Referente a Quantidade de Clusters

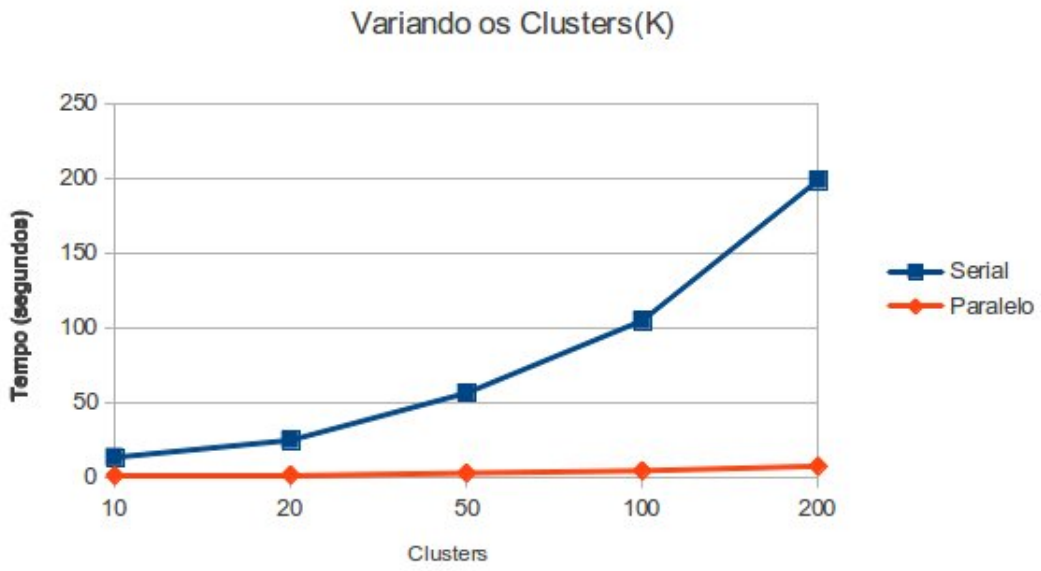

Figura 1. Gráfico comparando o tempo de execução entre o modelo seqüencial e paralelo

3.5. Variando a Quantidade de Iterações

Para este teste foi definida foram definidos alguns padrões, que serão listados abaixo:

- Resolução da Imagem: 1920x 1200
- Clusters(K): 50

N(Iterações)	Execução Seqüencial	Execução Paralela	SpeedUP
100	0m56.444s	0m2.617s	21.567
200	1m52.443s	0m2.854s	39.398
500	4m31.327s	0m3.583s	80.203
1000	9m3.976s	0m4.705s	115.617
2000	18m24.201s	0m7.130s	154.867

Tabela 2: Referente a Quantidade de Iterações

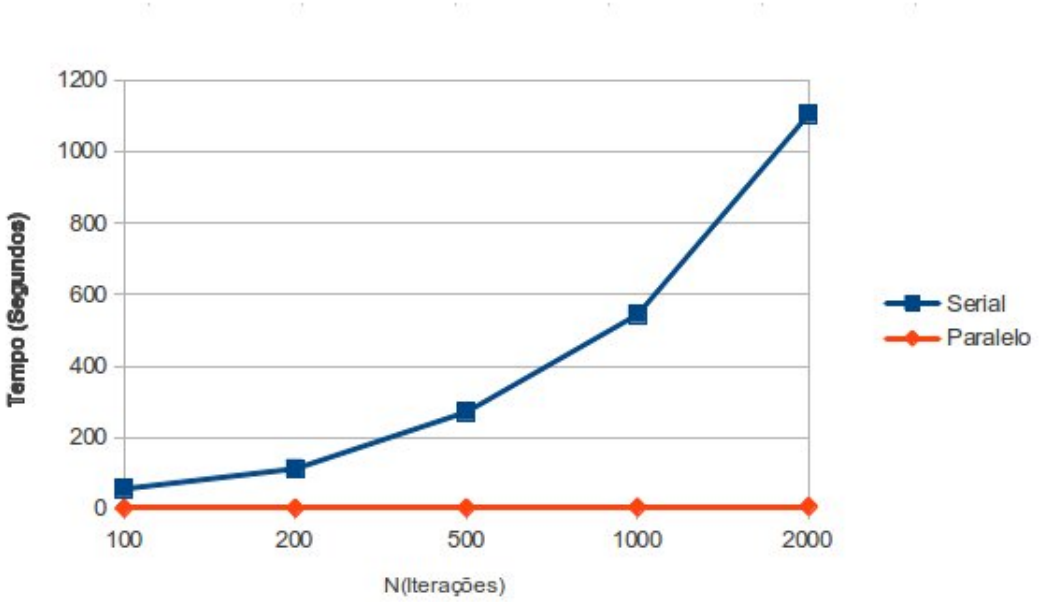

Figura 2. Gráfico comparando o tempo de execução entre o modelo seqüencial e paralelo

3.6. Variando o Tamanho da Imagem

Para este teste foi definida foram definidos alguns padrões, que serão listados abaixo:

- Iterações(N): 100
- Clusters(K): 50

Resolução	Execução Seqüencial	Execução Paralela	SpeedUP
160 x 160	0m0.181s	0m0.136s	1.33
480 x 480	0m1.418s	0m0.223s	6.359
912 x 800	0m4.311s	0m0.456s	9.454
1920 x 1200	0m13.279s	0m1.306s	10.168
4800 x 3200	1m24.102s	0m7.553s	11.135

Tabela 3: Referente a Resolução da imagem

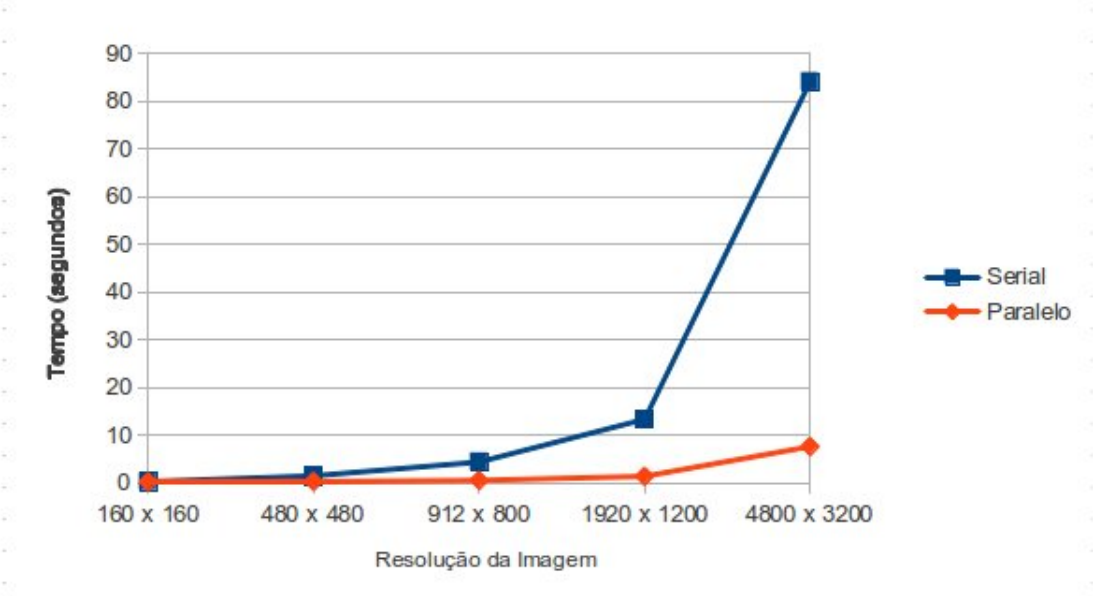

Figura 3. Gráfico comparando o tempo de execução entre o modelo seqüencial e paralelo

4. Conclusão

Neste relatório foi implementada uma versão paralela do Algoritmo de Clusterização K-medoid, com o intuito de comparar seu desempenho com a sua versão serial, em diferentes aspectos, utlizando a arquitetuta CUDA.

Depois dos vários testes realizados, nota-se a real eficiência do modelo paralelo em relação ao serial, neste caso, principalmente em relação a resolução da imagem.

Referências

BELTRAME, Walber Antônio Ramos; FONSECA, F. C. S. (2010). Aplicações práticas dos algoritmos de clusterização kmeans e bisecting k-means.