Computação Paralela
Algoritmo K-Means
Enunciado Prático 3

José Diogo Vieira
Departamento de Informática
Universidade do Minho
Braga,Portugal
pg50518

António Luís Fernandes
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg50229

***Abstract*—Dado um array inicial com valores aleatórios (x,y)), irá inicializar os K centroids com os primeiros K valores. Depois disso, vai associar cada ponto ao cluster que se encontrar mais próximo através da distância euclidiana. Por fim, depois de todos os pontos atribuídos, são calculados os novos valores dos centroids e repete-se o processo até não haver nenhum ponto a trocar de centroid. Neste relatório, iremos explicar a transição da implementação deste algoritmo sequencial para paralelo com recurso a CUDA.**

***Index Terms*—computação paralela, k-means, paralelismo, threads, otimização, CUDA, GPU**

I. Introdução

No âmbito de desenvolvimento do enunciado prático 3, foi-nos pedido novamente para otimizar o algoritmo k-means que tem sido desenvolvido nos trabalhos anteriores. Como opção de melhoramento, utilizamos o CUDA. O principal objetivo deste relatório é identificar como foi feita a implementação do algoritmo nesta plataforma e verificar, através de diferentes testes, a melhoria no tempo de execução quando comparado com o trabalho anterior (uso das primitivas OpenMp).

II. Versão Anterior - OpenMp

Em relação ao código da fase anterior, não foram feitas alterações uma vez que consideramos que a nossa abordagem já era satisfatória tanto em termos de execução como na ausência de data races, por isso preferimos abordar outras técnicas que serão melhor explicadas de seguida.

III. CUDA

A. o que é o CUDA?

CUDA trata-se de uma arquitetura de computação desenvolvida pela NVIDIA que permite que aplicações sejam executadas por vários núcleos de processamento da GPU.

B. K-means em CUDA

Nesta secção iremos abordar como foi feita a implementação do algoritmo k-means utilizando CUDA. Primeiramente, começamos pela inicialização dos vetores do algoritmo (com valores aleatórios entre 0 e 1), assim como a atribuição inicial para os valores dos centroids, esta parte mantém-se igual à fase anterior.

Inicialmente vamos alocar a memória na GPU para os vários elementos necessários para a execução do algoritmo:

- para os pontos X :
  `cudaMalloc((void **)&dX, bytes);`
- para os pontos Y :
  `cudaMalloc((void **)&dY, bytes);`
- para os pontos dos Centroids:
  `cudaMalloc((void **)&dC,bytesCentroids);`
- para o size dos centroids:
  `cudaMalloc((void **)&dSize, bytesInt);`
- para o somatórios dos pontos dos clusters:
  `cudaMalloc((void **)&dSum, bytesCentroids);`

e de seguida copiamos do host para o GPU, através da função cudaMemcpy, os várisos elementos necessários.

```
cudaMemcpy(dX, pointsX, bytes, \
cudaMemcpyHostToDevice);
cudaMemcpy(dY, pointsY, bytes, \
cudaMemcpyHostToDevice);
cudaMemcpy(dC, centroids, bytesCentroids, \
cudaMemcpyHostToDevice);
```

Com as estruturas devidamente prontas, podemos passar para a parte mais importante do nosso algoritmo. De modo a manter o mesmo comportamento da fase anterior, iremos manter o número de iterações do nosso algorimto em 20.

Como queremos que o size e o somatório seja iniciado a 0 para cada iteração, fazemos as seguintes atribuições, e as respetivas cópias para o device:

```
unsigned int size[K] = 0;
float sum[K * 2] = 0;
cudaMemcpy(dSize, size, bytesInt, \
cudaMemcpyHostToDevice);
 cudaMemcpy(dSum, sum, bytesCentroids,\
 cudaMemcpyHostToDevice);
```

De seguida, chamamos a nossa função do kernel:

```
kmeans<<<blocks, threads_block>>>\
(dX, dY, dC, dSize, dSum);
```

Esta será a função que ira calcular o cluster para cada ponto, somar e aumentar o tamanho do cluster a qual vai pertencer.

Inicialmente, começamos por pegar no indice do ponto:

```
int tid = blockIdx.x * blockDim.x \
+ threadIdx.x;
```

De seguida, iremos ver qual o cluster mais perto deste ponto. Esta parte é idêntica à da fase anterior, onde iteramos sobre os pontos de cada centroid e guardamos o indice da distância mais pequena.

Com o índice do cluster à qual vai pertencer descoberto, resta adicionar às estruturas do tamanho e do somatório, para tal usemos o seguinte código:

```
atomicAdd(&sum[X(min_index)], pointX[tid]);
atomicAdd(&sum[Y(min_index)], pointY[tid]);
atomicAdd(&size[min_index], 1);
```

de forma a evitar problemas de concorrência entre as várias threads, garantindo que estas serão feitas atomicamente. Esta não é a melhor maneira de resolver este problema, porque apesar do Atomic resolver as race conditions, isto vem com um custo, que é a performance. Para garantir que não há race, as threads têm de estar sincronizadas, o que leva um maior overhead por parte de cada thread.

Termina assim a função do kernel. Depois da execução da função de kernel, resta calcular os centroids. Para tal, é necessário copiar os resultados do Size e do Sum de volta para o Host:

```
cudaMemcpy(size, dSize, bytesInt,\
cudaMemcpyDeviceToHost);
cudaMemcpy(sum, dSum, bytesCentroids,\
cudaMemcpyDeviceToHost);
```

Já temos toda a informação necessária para o cáluculo dos centroids, que permanece igual às fases anteriores, dividir o somatório de cada cluster pelo o tamanho deste Com os novos centroids calculados, precisamos de voltar a enviar para o CPU:

```
cudaMemcpy(dC,centroids,\
bytesCentroids,cudaMemcpyHostToDevice);
```

C. Mudança na abordagem

De forma a melhorar a abordagem que explicamos anteriormente, mais precisamente melhorar a função de kernel. Vamos tentar evitar a utilização o atomicAdd, para o somatório e o size. De modo a diminuir o overhead de cada thread. O ideal seria arranjar alguma forma de conseguir fazer reduce, mas não conseguimos arranjar maneira de o fazer.

Desta forma, vamos redefinir a nossa função de kernel, para deixar de adicionar valores para o somatório e o size:

```
kmeans2<<<blocks, threads_block>>>\
(dX, dY, dC, dCluster);
```

Esta irá deixar de receber as estruturas de size e do somatório e passa a receber a estrutura que irá guardar qual o cluster a que cada ponto pertence.

Retiramos assim os CudaMalloc para o size e somatório e adicionamos a de cluster:

```
int bytesCluster = N * sizeof(int);
cudaMalloc((void **)&dCluster, bytesCluster);
```

Vamos fazer ainda outra alteração na função de kernel, esta para fazer uso da $shared\ memory$. Deste modo o acesso ao array dos centroids será feito mais rapidamente, para tal foram necessárias fazer as seguintes alterações:

- Declaração da variável partilhada

  ```
  __shared__ float sharedCentroids[K * 2];
  ```

- Atribuição dos pontos

  ```
  if (threadIdx.x < K * 2)
      sharedCentroids[threadIdx.x] = \
      centroids[threadIdx.x];
  ```

- Sincronização das threads no bloco

  ```
  __syncthreads();
  ```

A função para descobrir qual o cluster mais próximo permanece igual. Remove-se os atomicAdd utilzados para atualizar o Size e Sum. Adiciona-se a atribuição no cluster:

```
cluster[tid] = min_index;
```

Com esta alteração, passamos a ter que fazer o somatório e o incremento do size no host. Para tal, necessitamos dos pontos dos clusters:

```
cudaMemcpy(cluster,dCluster, bytesCluster,\
cudaMemcpyDeviceToHost);
```

E iremos fazer o incremento do size e o somatório sequencialmente. Assim como o cálculo dos centroids, que permance igual ao explicado anterirmente.

Esta abordagem, por um lado diminui o overhead na função de kernel, mas por outro lado aumenta significativamente o tamanho da informação que é copiada do Host para o Device, em cada iteração. Anteriormente, a informação copiada seria Size(K) + Sum(K*2), como K é o número de clusters, este número tem tendência a ser pequeno. No entanto, nesta abordagem a informação copiada será N, como o N é o número de pontos, este acaba por ser um valor elevado. i

IV. Desempenho da proposta implementada

A. Testes Realizados

Foram realizados testes para avaliar o desempenho da proposta implementada. Estes testes foram realizados no Cluster Search, com recurso ao comando nvprof. Todos os resultados são uma média de 5 execuções. Para a realização dos testes de performance do nosso código, fomos alterando o número de clusters inicial, começando com 4 e depois 8, 16, 32. Para

além disso, realizaram-se outros testes em que o número de pontos inicial varia.

Pontos	10 000 000	5 000 000	1 000 000	1000
4 clusters	1,13s	0,84s	0,3s	0,23s
8 clusters	1,40s	0,76s	0,43s	0,24s
16 clusters	2s	0,8s	0,39s	0,26s
32 clusters	2,3s	0,97s	0,3s	0,25s

TABLE I
RESULTADOS PARA 256 THREADS POR BLOCO

Nota : Os tempos de execução podem estar sujeitos a sobrecargas no cluster.

B. Análise dos Testes Realizados

Realizamos testes para 10 000 000, 5 000 000, 1 000 000 pontos com 4,8,16, 32 clusters.

Podemos verificar que para 10 000 000 ao aumentar o número de clusters, o tempo sobe proprocionalmente com o aumento do número de clusters.

No entanto, tanto para 5 000 000 pontos como para 1 000 000, esta subida já não é tão notória.

Como seria de esperar o aumento do número de pontos também provoca um aumento no tempo de execução.

Fizemos ainda a execução para 1000 para verificar como se comportaria se todos os valores pudessem ser acedidos na cache. Como podemos verificar os valore não são assim tão diferentes de 1 000 000, quando deveriam ser. Isto acontece porque maior parte do tempo é gasto na chamada ao CUDAMalloc, apesar dos tamanhos a alocar serem pequenos, o mais provável é, como esta é a primeira função de CUDA que utilizamos, deve estar a ser feito a inicialização do CUDA que por sua vez gasta algum tempo, que não é usado para a execução do algoritmo.

C. Comparação com OpenMP

Um fator determinante para o tempo de execução em openMP, passa pelo uso da primitiva $reduce$. Se conseguíssemos utilizar o comportamento desta em CUDA, o tempo diminuiria significativamente, pois não seria necessário efetuar a parte do somatório e do incremento do size sequencialmente. Sendo assim, não se consegue fazer uma comparação justa, visto que o no OpenMP implementamos o $reduce$ e no CUDA não.

D. Comparação com execução sequencial

Como seria de esperar, o tempo de execução em CUDA é bastante melhor do que sequencialmente, esta diferença é mais notória à medida que se aumenta o número de clusters.

E. Gráficos

De seguida, iremos ilustrar alguns gráficos de forma a visualizar o crescente tempo de execução dependendo número de clusters e para o mesmo número de clusters os diferentes tempos de execução dependendo do número de pontos iniciais.

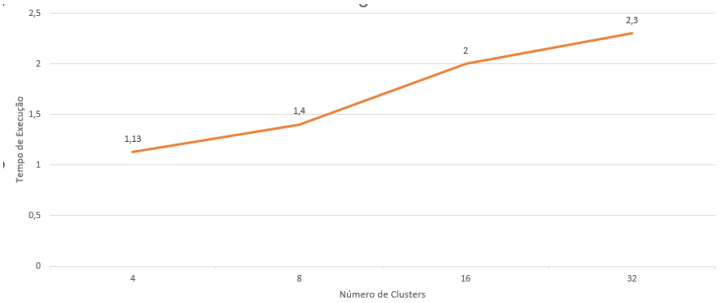

Fig. 1. Tempo de Execução em diferentes clusters para 10M de pontos

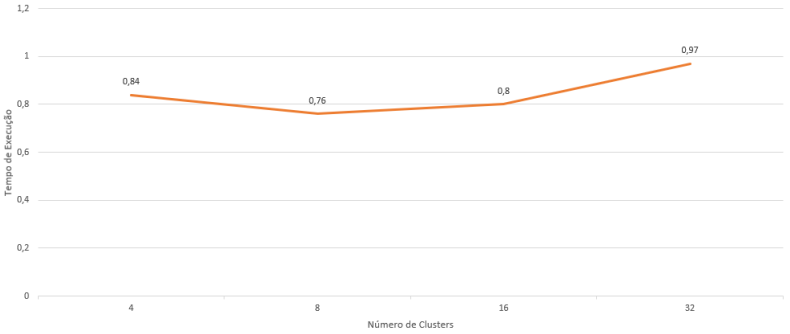

Fig. 2. Tempo de Execução em diferentes clusters para 5M de pontos

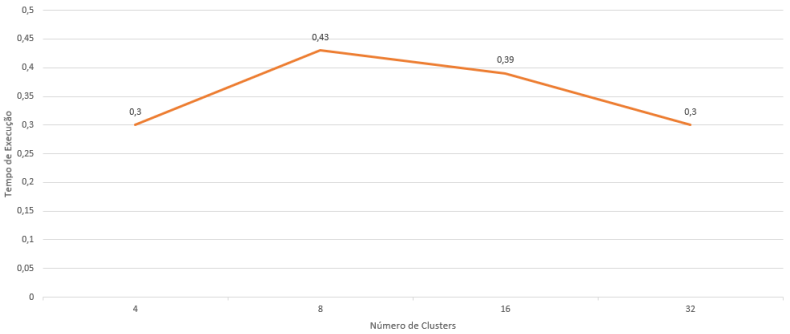

Fig. 3. Tempo de Execução em diferentes clusters para 1M de pontos

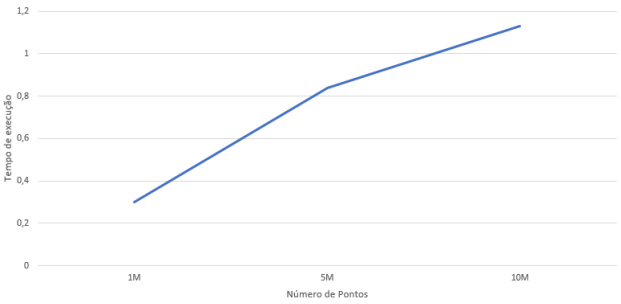

Fig. 4. Tempo de Execução para 4 clusters com diferente número de pontos iniciais (1M,5M,10M)

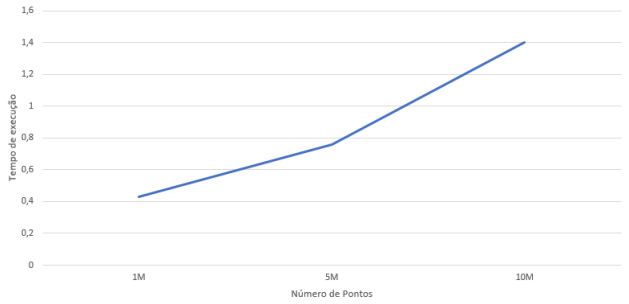

Fig. 5. Tempo de Execução para 8 clusters com diferente número de pontos iniciais (1M,5M,10M)

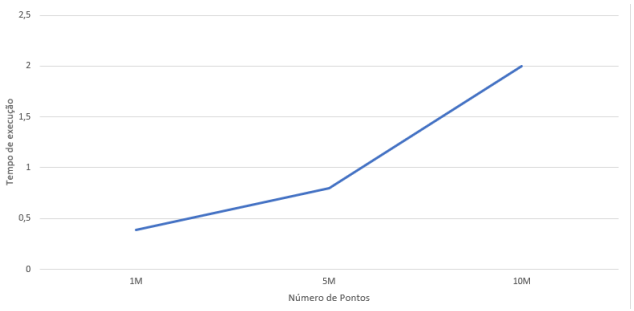

Fig. 6. Tempo de Execução para 16 clusters com diferente número de pontos iniciais (1M,5M,10M)

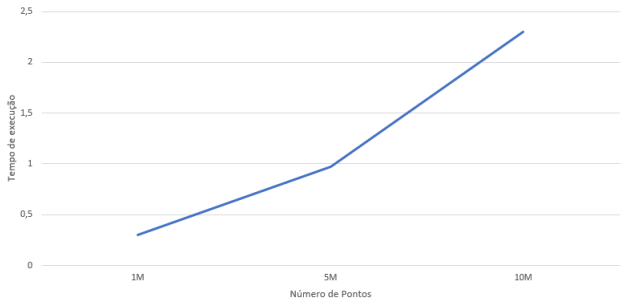

Fig. 7. Tempo de Execução para 32 clusters com diferente número de pontos iniciais (1M,5M,10M)

V. Conclusão

Dado por encerrado a realização deste trabalho, consideramos que o trabalho feito foi satisfatório e deu oportunidade para consolidar toda a matéria lecionada nas aulas práticas da UC.

A nossa maior dificuldade nesta terceira parte, foi como desenvolver o nosso código em CUDA, mais precisamente como adicionar os valores de sum e do size de uma forma mais eficiente, tentamos duas abordagens, mas sabemos que ainda poderia ser melhor se fossemos capazes de alguma forma fazer $reduce$.

Para trabalho futuro, gostaríamos de ter implementado em MPI para além de CUDA mas por restrições de tempo, optamos apenas pelo CUDA.