Computação Paralela - Trabalho Prático 3

1st João Vitor Vieira
Universidade do Minho
Braga, Portugal
PG50501

2nd José Pedro Magalhães
Universidade do Minho
Braga, Portugal
PG50528

***Abstract*—Este trabalho tem como objetivo corrigir e aprimorar a versão do algoritmo k-means com o decorrer de cada projeto com as táticas aprendidas ao longo da cadeira. Nesta fase em questão, o algoritmo irá ser otimizado com recurso a ambientes de programação, como o CUDA de forma a fazer uso da GPU e fazer mais uso do paralelismo.**

***Index Terms*—paralelismo, CUDA, MPI, k-means, threads, OpenMP**

I. Introduction

Na terceira fase do projeto, dando continuidade ao projeto anteriormente desenvolvido, o objetivo principal foi otimizar o algoritmo da melhor forma possível dado os recursos existentes, e, de seguida, avaliar e efetuar um estudo detalhado da escalabilidade da implementação.

Nesta fase, o grupo optou por abandonar a execução do algoritmo k-means em CPU que passou a ser executado na GPU de modo a criar uma solução mais escalável para grandes quantidades de pontos e/ou clusters, abandonando também consequentemente a utilização de OpenMP.

II. Implementação

Primeiramente, o grupo tentou fazer experiências rápidas no código desenvolvido no trabalho prático 2, de forma a aprimorar e dar mais uso às funcionalidades do OpenMP.

No entanto, rapidamente percebemos que o uso da GPU acabaria por se tornar uma mais valia no que toca à escalabilidade do algoritmo mesmo que isso significasse refazer o código na sua totalidade.

Um dos nossos principais focos foi tentar minimizar a quantidade de dados que é transferida entre a memória principal e a memória da GPU visto que este tipo de operações são extremamente demoradas.

Assim sendo, o código apresenta uma função principal que corre no CPU e dois GPU kernels, um deles responsável por atribuir pontos a um cluster(”kmeans_kernel”) e o outro responsável por atualizar as coordenadas dos clusters(”update_kernel”), uma vez que desta maneira apenas é necessário transferir dados quatro vezes em toda a execução do programa, duas no início para copiar as coordenadas dos pontos e dos clusters para a GPU e duas no final para copiar as coordenadas finais dos clusters assim como a quantidade de pontos em cada um para o CPU.

A. *Estruturas de Dados na GPU*

De forma a ser possível utilizar o GPU com os dados vindos da CPU, é necessário alocar memória para cada tipo de dados, sejam relativos tanto aos pontos como aos clusters. De forma a conter todos os dados, tanto principais como auxiliares, o grupo optou por utilizar cinco arrays alocados na memória da GPU.

- **gpu_points_coords** Array de tamanho ”número de pontos” * ”dimensão dos pontos(2 Dimensões neste caso)” que contém as coordenadas de todos os pontos.
- **gpu_clusters_coords** Array de tamanho ”número de clusters” * ”dimensão dos pontos(2 Dimensões neste caso)” que contém as coordenadas de todos os clusters.
- **gpu_sums** Array de tamanho ”número total de threads” * ”número de clusters” * ”dimensão dos pontos(2 Dimensões neste caso)” que contém as somas parciais dos pontos associados a cada cluster para cada thread.
- **gpu_counts** Array de tamanho ”número total de threads” * ”número de clusters” que contém o número de pontos em cada cluster por thread.
- **gpu_cluster_counts** Array de tamanho ”número de clusters” que contém o número total de pontos por cluster.

B. *Vetorização na GPU*

De modo a tentar melhorar ao máximo o tempo de execução optámos por utilizar um número total de threads inferior ao número de pontos, em que cada thread fica responsável por mais do que um ponto, por exemplo, se utilizarmos 3 threads no total então a primeira ficará responsável pelo primeiro ponto, pelo quarto, pelo sétimo... e assim sucessivamente.

O array gpu_sums tem como objetivo armazenar, para cada thread, a soma das coordenadas de cada um dos seus pontos para cada cluster.

Por outro lado o array gpu_counts, ao invés de calcular a soma das coordenadas, calcula o número de pontos associados à thread pertencentes a cada cluster. Ou seja, se tivermos 3 threads e 4 clusters então o array gpu_sums terá 3 secções de

tamanho 8 (uma vez que são pontos 2D) e o array gpu_counts terá 3 secções de tamanho 4.

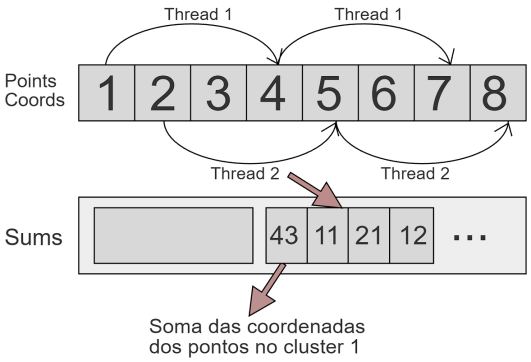

Fig. 1. Construção do array gpu_sums

Depois de preenchidos os arrays gpu_sums e gpu_counts cabe ao kernel ”update_kernel” iterar sobre cada um dos conjuntos em gpu_sums para calcular a soma total das coordenadas em cada cluster e posteriormente calcular as novas coordenadas através da média.

C. *Cálculo do número total de Threads*

O numero total de threads é um valor que divide a carga de trabalho entre o kernel ”kmeans_kernel” e o kernel ”update_kernel” isto porque é este valor que determina o comprimento do array gpu_sums, ou seja, quanto maior o número total de threads menor a carga de trabalho em cada thread do ”kmeans_kernel”.No entanto, como o comprimento do array gpu_sums será maior, maior será a carga de trabalho do kernel ”update_kernel”.

Tendo isto em conta e com o objetivo de distribuir a carga de trabalho pelos dois kernels o grupo chegou à seguinte fórmula:

$$NumTotalThreads = \sqrt{NumPontos * Dim * 2} \qquad (1)$$

Esta fórmula apresenta o melhor resultado não só ao nível do tempo de execução mas também é a fórmula que mais aproxima os tempos médios de execução dos dois kernels (10ms no ”kmeans_kernel” e 7ms no ”update_kernel”).

III. Escalabilidade

O grupo optou por fazer uma série de testes detalhados de modo a tentar perceber o nível de escalabilidade do programa realizado.

Todos os testes foram realizados no cluster utilizado durante as aulas práticas para várias quantidades de pontos e clusters e executando sempre 21 iterações tal como no segundo trabalho prático. Todos os resultados desta secção são uma média de pelo menos 3 tentativas e o tempo é calculado através do uso da função ”clock()” do módulo ”¡time.h¿” iniciando-se a contagem antes das alocações em GPU e terminando-se a contagem logo depois da libertação da memória da GPU.

O programa foi compilado com o ”nvcc” fazendo uso da Makefile disponibilizada para o guião 9 e correu com a ferramenta de ”profilling nvproff” o que nos ajuda a perceber quais as operações que mais influenciam o tempo de execução assim como saber quais as partes do código que devem ser melhoradas.

Importa salientar que em todos os testes a maior parte do tempo (mais de 80%) é dominada pelas operações de ”cudaMalloc” e ”cudaMemcpy”, operações essas que são inevitáveis para a execução do algoritmo.

A. *Em função dos Pontos*

Neste teste foram realizados testes com 1000, 100000, 1000000 e 10000000 pontos para 4 clusters.

Como é possível verificar pelo gráfico o aumento consecutivo do número de pontos não aumenta o tempo de execução de forma linear, o que é de esperar dado que cada thread do ”kmeans_kernel” trabalha sequencialmente com ”sqrt(numeroPontos * dim * 2)”, assim o aumento do tempo é logarítmico (ou semelhante).

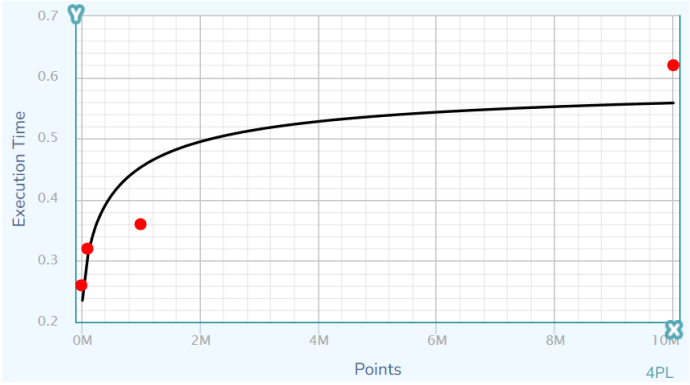

Fig. 2. Tempo de execução em função do número de pontos

B. *Em função dos Clusters*

Neste teste foram realizados testes com 4, 8, 16 e 32 clusters para 1000000 de pontos.

Como se pode observar na figura, o aumento do tempo em função dos clusters é linear uma vez que cada thread tem de iterar sobre todos os clusters sequencialmente.

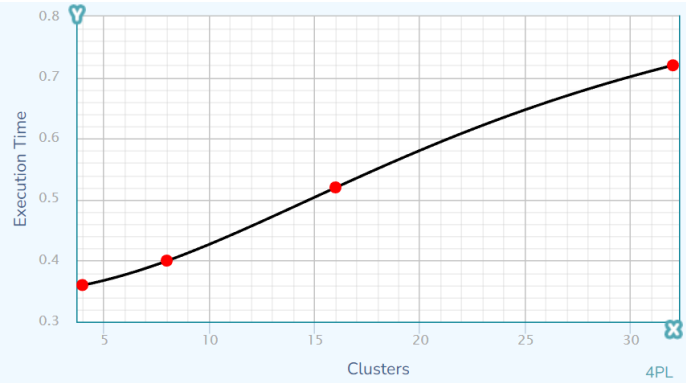

Fig. 3. Tempo de execução em função do número de clusters

IV. Resultados

A. *Resultados obtidos*

Para além da influência dos pontos dos clusters para a análise de escalabilidade o grupo optou também por testar todas as combinações dos valores previamente utilizados como se mostra na seguinte tabela.

	4 Clusters	8 Clusters	16 Clusters	32 Clusters
1000 Pontos	0.20s	0.24s	0.26s	0.28s
100000 Pontos	0.32s	0.38s	0.43s	0.48s
1000000 Pontos	0.36s	0.40s	0.52s	0.72s
10000000 Pontos	0.62s	0.82s	1.20s	1.83s

Importa também comentar que com a ajuda do profiller(nvprof) podemos também concluir que em média cerca de 80% do tempo de execução corresponde à sincronização entre o CPU e a GPU, à alocação de memória na GPU e à posterior transferência de informação para a memória da GPU nomeadamente no que diz respeito ao array com as coordenadas dos pontos sendo este de tamanho considerável.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.32%  230.28ms        21  10.966ms  10.936ms  11.005ms  kmeans_kernel(int*, fl
                   36.50%  146.66ms        21  6.9837ms  6.9430ms  7.0335ms  update_kernel(int*, in
                    6.18%  24.810ms         2  12.405ms     896ns  24.809ms  [CUDA memcpy HtoD]
                    0.00%  9.4730us         2  4.7360us  4.3200us  5.1530us  [CUDA memset]
                    0.00%  3.5830us         2  1.7910us  1.7270us  1.8560us  [CUDA memcpy DtoH]
      API calls:   63.98%  376.69ms         1  376.69ms  376.69ms  376.69ms  cudaDeviceSynchronize
                   30.13%  177.40ms         5  35.479ms  4.3550us  177.23ms  cudaMalloc
                    4.30%  25.299ms         4  6.3248ms  17.532us  25.149ms  cudaMemcpy
                    1.44%  8.4606ms         5  1.6921ms  5.4370us  8.1145ms  cudaFree
```

Fig. 4. Resultados de profilling para 4 clusters e 10000000 de pontos

Apenas a restante percentagem de tempo corresponde à execução dos kernels em GPU, execução essa que foi exaustivamente trabalhada pelo grupo de modo a obter os melhores resultados possíveis.

B. *Comparação com o TP2*

No trabalho prático 2 o grupo fez uso de de OpenMP onde realizámos testes para 4, 8, 16 e 32 clusters com 1000000 de pontos com 16 threads tendo sido obtidos os seguintes resultados:

	4 Clusters	8 Clusters	16 Clusters	32 Clusters
1000 Pontos	0.007s	0.011s	0.008s	0.013s
100000 Pontos	0.05s	0.06s	0.07s	0.23s
1000000 Pontos	0.34s	0.44s	0.61s	0.87s
10000000 Pontos	2.97s	3.98s	5.06s	6.86s

Como se pode ver pela tabela assim como na imagem seguinte o tempo de execução aumenta linearmente com o número de pontos o que mostra que esta terceira fase do projeto apresenta uma enorme vantagem em relação à fase anterior.

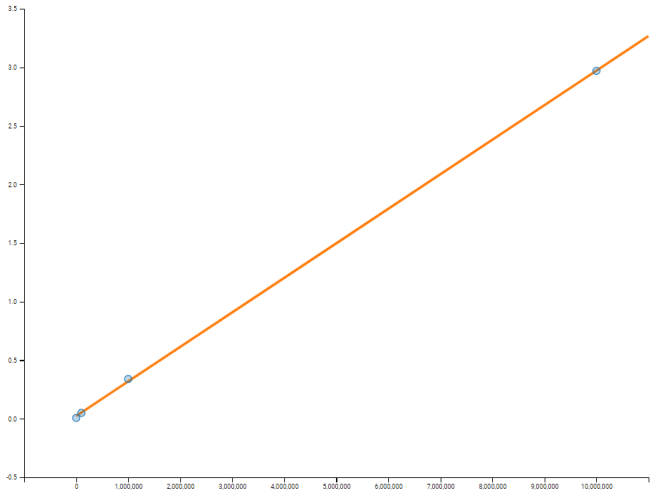

Fig. 5. Tempo de execução em função do número de pontos (TP2)

Importa deixar claro que embora os tempos de execução desta fase aumentem de forma logarítmica não é esperado que este comportamento se mantenha indefinidamente visto que a GPU tem memória limitada.

V. Conclusão

A terceira fase do trabalho prático de Computação Paralela apresentou-se como uma oportunidade para o grupo aplicar e ampliar os conhecimentos adquiridos em sala de aula. O desafio proposto foi um incentivo para o grupo desenvolver as suas habilidades no uso de técnicas de computação paralela, permitindo-nos alcançar resultados bastante satisfatórios.

Os baixos tempos de execução obtidos foram o reflexo do esforço e dedicação do grupo em alcançar o melhor desempenho possível. Além disso, a grande quantidade de conhecimento adquirido ao longo desta fase foi um aspeto

muito valorizado pelo grupo, pois permitiu-nos avançar significativamente no nosso desenvolvimento como programadores.

Em conclusão, esta fase foi uma experiência altamente enriquecedora para o grupo, permitindo-nos aplicar os nossos conhecimentos teóricos em contexto prático e desenvolver as nossos habilidades no uso de técnicas paralelas. O grupo mostra-se bastante satisfeito com os resultados obtidos e ansioso por continuar a aprender e a desenvolver as nossas capacidades no futuro.

VI. Referências

1 - https://www.openmp.org/resources/refguides/

2 - https://docs.nvidia.com/cuda/