Trabalho Prático 3

Computação Paralela

1st Duarte Augusto Rodrigues Lucas - PG50345
Universidade do Minho
Mestrado em Engenharia Informática
Santo Tirso, Portugal
pg50345@alunos.uminho.pt

2nd Tiago Fernandes Ribeiro - PG50779
Universidade do Minho
Mestrado em Engenharia Informática
Barcelos, Portugal
pg50779@alunos.uminho.pt

***Abstract*—Este documento é relativo ao trabalho prático 3 da unidade curricular de Computação Paralela, que tem como objetivo a exploração de paralelismo no programa criado nos trabalhos anteriores, utilizando ambientes de programação paralela alternativos de modo a diminuir o seu tempo de execução.**

***Index Terms*—computação paralela, CUDA, paralelismo, linguagem C, algoritmo k-means**

I. Introdução

Este projeto tem como objetivo corrigir/otimizar a versão paralela do algoritmo, já desenvolvido nas fases anteriores, em OpenMP ou explorar outras plataformas computacionais/ambientes de programação para exploração de paralelismo, recorrendo a librarias como o CUDA ou MPI.

Esta atualização do código foi desenvolvida na linguagem de programação C, com o auxílio da biblioteca de programação paralela CUDA, *Compute Unified Device Architecture*, desenvolvida pela NVIDIA, sendo que consiste numa nova versão do programa desenvolvido nos trabalhos práticos anteriores. O programa implementa o algoritmo k-means simples, com base no algoritmo de Lloyd. Este algoritmo classifica um conjunto de amostras (pontos) em k grupos (clusters).

Mantendo sempre a filosofia do algoritmo original, o objetivo passa por minimizar o tempo total de execução do algoritmo, utilizando técnicas de otimização de código, recorrendo a técnicas e ferramentas de análise de código, sendo que nesta fase se foca mais na utilização de bibliotecas de programação paralela alternativas.

II. Alterações no código

No desenrolar desta fase, o grupo de trabalho, após uma discussão sobre qual o melhor método a seguir nesta terceira e última fase do projeto, concluiu que seria de melhor proveito e com uma possível melhor resolução do problema utilizar a livraria CUDA.

A livraria CUDA é um conjunto de ferramentas de programação e plataforma de desenvolvimento desenvolvido pela NVIDIA para aproveitar ao máximo o poder de processamento de GPUs. Esta fornece uma série de bibliotecas e APIs que facilitam o desenvolvimento de aplicações que utilizem paralelismo.

Inicialmente foi necessário analisar que alterações iriam ser necessárias de modo a dar o devido uso das funcionalidades disponibilizadas pelo CUDA. Deste modo, começamos por identificar que funções iriam correr na GPU (*device*) e na CPU (*host*). A GPU é um dispositivo de tratamento especializado que é projetado para acelerar o processamento gráfico, mas também pode ser usada para acelerar o tratamento de dados gerais. A CPU, por outro lado, é o principal dispositivo de processamento de um computador e é responsável por executar a maioria das tarefas do sistema. A escolha de que funções devem correr na GPU ou na CPU revela-se de extrema importância a nível do tempo de execução pois a utilização da GPU demonstra uma vantagem quando na função pode ser utilizado o paralelismo, caso contrário a CPU revela-se mais adequada. Deste modo, concluímos que as funções atribuiCluster e calculaCentroid são adequadas para correr na GPU, todas as outras deverão correr no CPU.

Nesta parte final do projeto desenvolvido, de modo a adaptar o código à livraria que está a ser utilizada, foi necessário criar estruturas de dados. A estrutura *sumThread*, contém o somatório das coordenadas x, o somatório das coordenadas y e o número de pontos que a *thread* correspondente analisou. A *struct Point*, é a estrutura que armazena as coordenadas x e y de um ponto. Foram ainda retiradas todas as variáveis que anteriormente estavam declaradas, passando a ser variáveis privadas declaradas na função main, que por sua vez passará como argumentos a outras funções.

A. Função calculaCluster

Esta função descobre o cluster mais próximo da amostra e efetua o preenchimento da *struct sumThreads*. É uma função que corre no *device*, isto é na GPU, e recebe como argumentos o *array* de pontos, o *array* de centroids, o *array* da estrutura do somatório das *threads*, o número de pontos, o número de clusters e o número total de *threads*. Este kernel é executado com o número de blocos e com o número de *threads* por blocos, que foram recebidos pelo input.

Inicialmente é calculado o id da thread a correr o kernel, depois é feito um ciclo *for* que percorre todos os pontos e que calcula a distância para o primeiro cluster. Ainda dentro do ciclo *for* é realizado outro ciclo que percorre os vários clusters para comparar a distancia da amostra ao seu centroid. Após

o fim do ciclo *for* interior terminar, a *thread* soma ao cluster mais próximo as coordenadas x e y no seu espaço designado no *array* dos somatórios das *threads* e incrementa o número de pontos analisados por ela, sendo que depois irá analisar outro ponto. Este novo ponto a analisar estará no índice do último ponto analisado mais o número total de *threads* a correr o kernel.

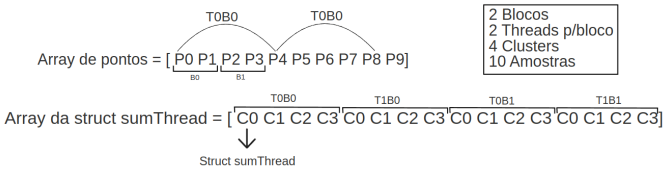

Fig. 1. Exemplo de funcionamento da função calculaCluster

Neste exemplo a *thread* 0 do bloco 0, calcula o melhor centroid para os índices do *array* de pontos 0, 4 e 8.

A *struct sumThread* presente no primeiro índice do *array* dos somatórios das *threads* contém o somatório das coordenadas x e y dos pontos e o número de pontos que a *thread* 0 calculou que estavam perto do cluster 0.

B. Função calculaCentroid

É a função que calcula o centroid de cada cluster no *device*, isto é na GPU. Recebe como argumentos o *array* dos centroids, o *array* da estrutura dos somatórios das *threads*, o *array* de inteiros para o tamanho de cada *cluster*, o número de clusters e o número total de *threads*. Este kernel é executado com apenas 1 bloco com tantas *threads* quanto o número de clusters.

Começa por calcular o índice de cada *thread*, sendo que existem tantas threads quanto o número de clusters, e em seguida reinicializa as variáveis dos centroids. Após isto, executa um ciclo *for* que percorre o *array sumThreads* e efetua o somatório dos somatórios efetuados por cada *thread* no kernel anterior, acumulando estes valores nas variáveis dos centroids e no *array* dos tamanhos dos clusters. Cada thread soma os valores relativos a apenas um cluster, o que impede a existência de *data races*. No fim do ciclo *for* é feito o cálculo do novo centroid, sendo para cada cluster realizada a divisão do somatório, presente nas variáveis das coordenadas, pelo tamanho desse mesmo cluster.

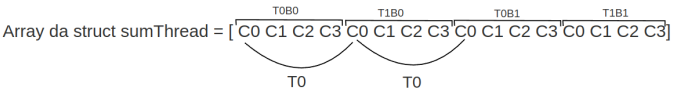

Fig. 2. Exemplo de execução da função calculaCentroids

Como é visível no exemplo, são criadas tantas *threads* quanto o número de clusters e cada uma efetua o somatório dos somatórios das coordenadas calculadas por cada *thread* no kernel anterior. Com estes somatórios são calculados os novos centroids. Os valores em cima do *array*, indicam por qual *thread* é que os somatório foram calculados no kernel anterior.

C. Função inicializa

Função responsável pela inicialização da lista de pontos, de forma aleatória, e também pela inicialização dos *arrays* dos centroids, dos somatórios de cada *thread* e dos tamanhos dos clusters. Ela recebe cada um destes *arrays*, mais informações sobre o número de pontos, clusters e número total de *threads* a executar. Esta função é executada no *host*, ou seja, no CPU. Percorre o *array* dos pontos e efetua o preenchimento deste de forma aleatória, com a função *srand* que recebe uma *seed* para gerar sempre os mesmos valores. De seguida, preenche o *array* dos centroids com os valores dos primeiros número de clusters pontos. Após isto, inicializa os *arrays* do somatório das threads e dos tamanhos dos clusters a 0.

D. Função main

É a função responsável pela execução de todo o programa, começando por receber no input do *stdin* o valor das variáveis do número de pontos, o número de clusters, o número de blocos e o número de *threads* por bloco. Com esta informação é realizado o cálculo do número total de threads a executar, multiplicando o número de blocos pelo número de *threads* por bloco.

Em seguida, são criadas as variáveis presentes no *host*, sendo estas o *array* de pontos, o *array* dos centroids, o *array* com a estrutura dos somatórios de cada *thread* e o *array* com os tamanhos de cada cluster. Depois, é alocada memória para todas as variáveis criadas, fazendo uso da função *malloc*. Por fim, são inicializadas as variáveis da GPU: o *array* de pontos, o *array* de centroids, *array* com a estrutura dos somatórios de cada *thread* e o *array* com os tamanhos de cada *thread*. Sendo também necessário alocar memória para estas através da função *cudaMalloc*, que destina espaços na memória da GPU para estas informações. Só depois de receber as variáveis do input, inicializar e alocar memória para as variáveis do CPU e da GPU, se dá inicio à execução do algoritmo K-means. Primeiro, com a função *inicializa* é realizada a inicialização da lista de pontos de forma aleatória, a inicialização dos *arrays* dos centroids, do somatório de cada thread e dos tamanhos dos clusters, variáveis presentes no CPU . Depois, através da função *cudaMemcpy* é copiada a memória, onde estão cada uma destas variáveis, da CPU para a GPU. Só depois destes passos é que se realiza o *loop* de modo a concluir o algoritmo, em que em cada iteração são realizados os cálculos da distância de cada ponto aos *centroids* de cada cluster, na GPU utilizando a função *calculaCluster*, e em seguida é feito o cálculo dos novos centroids com a função *calculaCentroids*. Este *loop* não é executado até aos clusters convergirem, mas sim até chegar a um limite de iterações definido (21 iterações). Após a conclusão do *loop*, é utilizada a função *cudaMemcpy* para copiar os *arrays size* e *centroids* da GPU para as respetivas variáveis do CPU. Por fim, libertamos a memória da GPU, fazendo uso da função

cudaFree e libertamos a memória do CPU utilizando a função *free*.

III. Testes

Neste tópico são apresentados os resultados obtidos ao executar o programa e serão comparados com a execução do algoritmo em CPU de maneira sequencial e a versão em OpenMP. Além disso, serão realizados testes de escalabilidade para verificar o comportamento do programa em diferentes tamanhos de conjunto de dados. O objetivo desta secção é avaliar o desempenho do programa e verificar se ele é capaz de aproveitar ao máximo o poder de processamento da GPU.

A. Especificações da máquina utilizada para os testes

Para efeitos de teste, foi utilizada a máquina pessoal de um dos integrantes do grupo, tendo esta as seguintes especificações de hardware:

- Processador: Intel Core i5-8265U
- Placa gráfica (Integrada): Intel UHD Graphics 620
- Placa gráfica (Dedicada): NVIDIA GeForce MX110
- Memória RAM: 12GB

TABLE I
Especificações da GPU

	Tamanho/Velocidade
VRAM	2 GB
Frequência da memória	2446 MHz
Tamanho do bus de memória	64 bits
Largura de banda da memória	40.1 Gb/s
Número de SMM's	3

TABLE II
Tamanho das várias hierarquias de memória da GPU

	Tamanho
L1	64 KB p/ SMM
L2	1024 KB
VRAM	2 GB

TABLE III
Informações da arquitetura multiprocessador da GPU

	Tamanho
Nº máximo de *threads* por bloco	1024
Tamanho de um *Warp*	32
Nº máximo de blocos residentes p/ multi-processador	32
Nº máximo de *warps* residentes p/ multi-processador	64
Nº máximo de *threads* residentes p/ multi-processador	2048
Nº de registos regulares de 32-bits p/ multi-processador	64 K
Nº máximo de registos de 32-bits p/ bloco de *thread*	64 K
Nº máximo de registos regulares de 32-bits p/ *thread*	255
Quantidade de memória partilhada p/ multi- processador (de toda a memória partilhada + cache L1, quando aplicável)	96 KiB
Quantidade máxima de memória partilhada p/ bloco de *thread*	48 KiB
Quantidade de memória local p/ *thread*	512 KiB

A GPU tem a versão da capacidade de computação 6.1 e segue a micro-arquitetura Pascal.

TABLE IV
Especificações do CPU

	Número/Velocidade
Cores	4
Threads	8
Frequência	1.6 GHz
Frequência máxima	3.9 GHz

TABLE V
Tamanho das várias hierarquias de memória

	Tamanho
Cache L1	256 KiB
Cache L2	1 MiB
Cache L3	6 MiB
RAM	8 GiB + 4 GiB

As especificações da máquina utilizada para os testes do programa são importantes, nomeadamente o tamanho das várias caches, pois estas ditam os valores das várias variáveis a utilizar durante a execução dos testes.

B. Métricas a utilizar

Para a avaliar o desempenho do modelo foram identificadas várias métricas que são importantes para este efeito. Em primeiro lugar, sendo esta uma das métricas mais relevantes, o tempo de execução, pois ele permite ajudar a determinar se a execução do algoritmo está a ser eficiente e se existe algum *bottleneck* no programa. O *profiling* do código também é importante, pois permite visualizar quais as partes onde o programa passa mais tempo, permitindo também identificar *bottlenecks*. Existem outras métricas importantes que podiam ser benéficas para a avaliação do desempenho do código como o número de cache misses, que pode indicar se o programa está a ser penalizado por ter de efetuar acessos às camadas mais profundas da hierarquia de memória, porém o acesso a estas métricas não é disponibilizado por parte da GPU. Desta forma, as métricas utilizadas são fundamentais para avaliar o desempenho do modelo.

IV. Resultados

Nos testes efetuados foram utilizados números de blocos e de *threads* por bloco que se adequem ao tamanho dos dados de *input* e que maximizem a taxa de ocupação, ou seja maximizar o número de *threads* ativas ao mesmo tempo. Nas execuções do programa são utilizados um número de *threads* adequado à amostra (1), onde o número de *threads* é aproximadamente a raiz quadrada do número de amostras, outro número de threads que maximize a taxa de ocupação (3) e um número de threads que equilibre estes dois (2). Para cada um destes modelos foram testadas várias combinações do número de blocos e do número de *threads* por bloco, de modo a que respeitem sempre o número total de *threads* sugerido pelo modelo, sendo que as tabelas dos resultados apenas são apresentadas as melhores combinações para cada um dos modelos.

Os resultados das tabelas dos tempos de execução são calculados a partir de uma média de 10 execuções do programa. Foram testados quatro valores diferentes do número

de clusters: 4 e 32. Quanto ao tamanho do *array* de pontos foram testados três valores: Um valor na ordem dos 8 mil pontos que faz com que o *array* possa ser inserido na sua totalidade no nível de cache L1, outro valor na ordem dos 100 mil pontos que torna possível a inserção do *array* na cache de nível L2 e por fim um *array* de 10 milhões de amostras que impossibilita a inserção de todos os dados nos níveis superiores da cache e obriga o programa a utilizar níveis mais profundos da hierarquia de memória para aceder aos dados.

TABLE VI
TEMPOS DE EXECUÇÃO EM SEGUNDOS COM 4 CLUSTERS

Total de amostras \Nº total de threads	**(1)**	**(2)**	**(3)**
8 mil (∼64KB)	0.098	0.175	0.240
100 mil (∼800KB)	0.116	0.165	0.166
10 milhões (∼8MB)	1.284	1.410	1.385

Para 4 clusters, o primeiro modelo, onde o número total de *threads* a executar é igual à raiz quadrada do total de amostras, é o que se destaca mais.

TABLE VII
TEMPOS DE EXECUÇÃO EM SEGUNDOS COM 32 CLUSTERS

Tamanho da amostra \Nº total de threads	**(1)**	**(2)**	**(3)**
8 mil (∼64KB)	0.106	0.175	0.242
100 mil (∼800KB)	0.155	0.192	0.194
10 milhões (∼8MB)	2.014	2.332	2.213

Com 32 clusters, o modelo (1) continua destacadamente com os melhores tempos de execução. Nos vários testes, o tamanho dos *arrays* a utilizar é adequado a cada um nos níveis da hierarquia de memória da máquina (L1 e L2). É de notar que cada *float* ocupa 4 bytes, sendo que a *struct* onde são armazenados os pontos contém dois *floats* (coordenadas x e y). Ao alterar o tamanho do *array* para os diferentes níveis da cache irá fazer com que o número de cache misses diminua e o programa não tenha de ir tantas vezes a níveis mais profundos da memória para obter os dados necessários à execução do mesmo.

Relembrando os resultados dos testes, com 10 milhões de amostras, executados na versão do programa desenvolvido no trabalho prático dois:

TABLE VIII
COMPARAÇÃO DO TEMPO DE EXECUÇÃO EM SEGUNDOS E SPEEDUP COM AS OUTRAS VERSÕES DESENVOLVIDAS

	4 clusters	**Speedup**	**32 clusters**	**Speedup**
Sequencial	2.43	1	13.39	1
OpemMP (16 *threads*)	0.96	2.53	1.68	7.97
CUDA versão (1)	1.28	1.90	2.01	6.66

Com a observação da tabela, é possível concluir que a versão do programa em OpenMP atinge melhores tempos de execução e, consequentemente, tem um *speedup* maior em relação à versão sequencial. A versão desenvolvida nesta fase do trabalho prático peca por alguns motivos, que podem ser visualizados na seguinte figura.

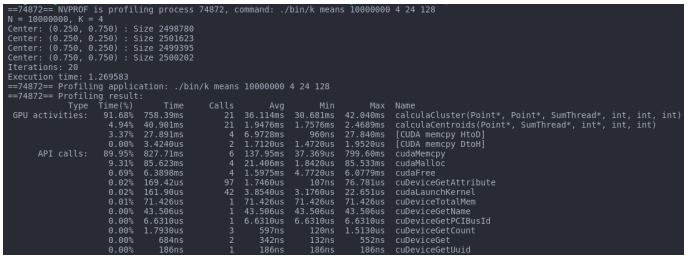

Fig. 3. Profiling do código desenvolvido

O grupo de trabalho, de modo a analisar e melhorar a *performance* do programa, faz uso da ferramenta de *profiling* do *toolkit* do CUDA, o *nvprof*. O profiling é uma técnica utilizada para medir o desempenho de um programa e identificar quais partes dele estão a consumir mais recursos. Como é possível visualizar na figura a cima, verifica-se que o programa desenvolvido consome grande parte do tempo executado em duas tarefas, a função *cudaMemcpy* e a *calculaCluster*, sendo que a *cudaMemcpy* a que mais demorada. Esta ocupação do tempo por parte da *cudaMemcpy* deve-se ao facto de ser necessário passar todas as estruturas criadas no início do algoritmo, do *host* para o *device*. No fim da execução do algoritmo, também é necessário passar o tamanho dos clusters e os centroids para o CPU. Em relação ao *calculaCluster*, este também demonstra que ocupa boa parte do tempo total, devendo-se ao facto desta ser a função responsável por percorrer os pontos e por cada ponto, percorrer os clusters todos e efetuar o cálculo da distância do ponto ao centroid do respetivo cluster.

V. CONCLUSÃO

Este terceiro trabalho prático permitiu-nos consolidar os temas abordados no trabalho anterior bem como os novos temas abordados para este trabalho, de entre estes a programação com recurso a aceleradores gráficos, mais concretamente a biblioteca CUDA. Para isto, foi necessário pesquisar sobre estes temas para obter um conhecimento mais pormenorizado sobre as funcionalidades desta livraria, para além do conhecimento já obtido nas aulas teóricas e práticas, de modo a obter o melhor resultado possível. Com isto, concluímos também que todo o trabalho permitiu-nos adquirir um maior conhecimento e experiência sobre a manutenção e construção do código de modo a tornar o programa mais eficiente, conhecimento este que se tornará útil em trabalhos futuros e na nossa vida profissional.

REFERENCES

[1] datasciencelab, ”Clustering With K-Means in Python”, https://datasciencelab.wordpress.com/tag/lloyds-algorithm/
[2] OpenMP ARB, ”OpenMP 4.0 API C/C++ Syntax Quick Reference Card”, https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf
[3] Wikipedia, ”Cuda”, https://en.wikipedia.org/wiki/CUDA
[4] Nvidia, ”CUDA Toolkit - Develop, Optimize and Deploy GPU-Accelerated Apps”, https://developer.nvidia.com/cuda-toolkit
[5] Nvidia, ”An Even Easier Introduction to CUDA”, https://developer.nvidia.com/blog/even-easier-introduction-cuda/