Computação Paralela
Algoritmo k-Means - Paralelismo

1st Catarina Martins
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50289@alunos.uminho.pt

2nd Joaquim Roque
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50502@alunos.uminho.pt

***Resumo*—O algoritmo *k-Means* é um método de agrupamento de pontos que subdivide um conjunto de N amostras em k *clusters*. É um processo iterativo em que a cada amostra é-lhe atribuída um *cluster*, com base na menor distância euclidiana ao centro do mesmo (referido como centroide). O processo iterativo termina quando se verifica que o algoritmo convergiu, ou seja, nenhum ponto mudou de *cluster*.**

Index Terms*—otimização, *performance*, CUDA, paralelismo, *kernel*, *cluster

I. Introdução

Para a fase final da componente prática da Unidade Curricular de Computação Paralela, é-nos proposto que se desenvolva um programa onde se explore eficientemente o paralelismo, tendo por base o trabalho realizado na fase anterior, sendo dada a liberdade de escolha do ambiente de programação a usar para esse efeito.

II. Formulação

Para a realização do trabalho, o grupo começou por analisar de que forma poderia explorar paralelismo a aplicar no algoritmo em questão, tendo em consideração os três ambientes de programação paralela abordados (OpenMP, MPI, CUDA). Inicialmente foi considerada a hipótese de melhorar a implementação em OpenMP da versão anterior, recorrendo a outras diretivas e conceitos da API, nomeadamente *tasks*. No entanto, não foi identificado uma forma aparente de tirar proveito da sua utilização, pelo que esta opção foi descartada. O uso de MPI foi também desde logo rejeitado, dado que consideramos que o *overhead* do lançamento de novos processos e da comunicação entre os mesmos seria prejudicial. Dessa forma, tomou-se a decisão de recorrer à plataforma de programação em GPUs da NVidia (i.e. CUDA). Um dos casos de uso da mesma é proporcionar uma forma eficiente de efetuar uma grande taxa de cálculos matemáticos (*number crunching*), pelo que se apresenta como adequada para o problema em questão.

III. Implementação

Para a implementação do algoritmo, foram desenvolvidas duas *kernels*, cujas assinaturas se apresentam em seguida:

```
__global__ static
void compute_partial_centroids_kernel(
DeviceVector<Sample> sv,
DeviceVector<Sample> centroids,
DeviceVector<Sample> centroids_accum,
DeviceVector<size_t> cluster_sizes_accum
);

__global__ static
void reduce_centroids_kernel(
DeviceVector<Sample> centroids,
DeviceVector<size_t> cluster_sizes,
DeviceVector<Sample> centroids_accum,
DeviceVector<size_t> cluster_sizes_accum
);
```

Esta abordagem elimina a necessidade de sincronização e/ou exclusão mútua entre *threads* (continua a ser necessário sincronizar as *kernels*, uma vez que há *overlap* entre os dados sobre os quais as mesmas operam). Isto revela-se fundamental para o desempenho do programa, uma vez que a partir de implementações anteriores testadas pelo grupo se verificou que as funções de controlo de concorrência da API do CUDA revelam-se extremamente ineficientes, sobretudo para um elevado número de fios de execução.

Desta forma, garante-se a correção do algoritmo replicando os vetores de acumulação do cálculo dos novos centroides pelas várias *threads*. Por outras palavras, cada *thread* acessa a uma zona efetivamente privada dos vetores (i.e. *accumulator_centroids* e *accumulator_cluster_sizes*) onde serão acumulados resultados parciais do cálculo dos próximos centroides, a usar na iteração seguinte. A segunda *kernel* encarrega-se depois de reduzir os valores parcialmente calculados para os novos centroides. Esta lógica é muito semelhante ao que acontece com as diretivas de redução do OpenMP.

De notar que entre as chamadas às *kernels* não existe nenhuma operação no *host*. A alocação de memória na GPU e a cópia de dados para a mesma são feitas apenas no início do programa, evitando *overhead* de constante comunicação entre *host* e *device*. O número de chamadas a estas funções é, portanto, constante e independente do *input*. Apesar disso, o seu tempo de execução não é necessariamente constante, uma vez que a quantidade de *bytes* copiados depende inevitavelmente do tamanho do *input*.

No que toca à distribuição da carga na primeira *kernel*, cada *thread* tem um identificador único id, calculado com base no índice bloco a que pertence, a dimensão de cada bloco e o identificador do fio de execução dentro do bloco. Dessa forma, cada *thread* calcula o novo *cluster* de cada ponto cujo índice i no vetor de amostras satisfaça a igualdade $i \ \% \ total\_number\_of\_threads = id$. Com isto, é possível otimizar o processo de iteração sobre este mesmo vetor, usando como passo do ciclo o número total de *threads*. Apesar deste método dar a entender que os acessos à memória são não contíguos, a verdade é que este modelo é ideal para programação no ambiente CUDA, uma vez que o bloco de memória acedido por um conjunto de *threads* (i.e. *warp*) é contíguo, devido à forma como os *ids* das *threads* são calculados, bem como a regularidade nos acessos dentro do dito *warp*. Este fenómeno tem o nome de *coalescing*.

Já para a *kernel* de redução são invocadas tantas *threads* quanto o número de *clusters*. Em teoria, para números de *clusters* baixos e elevado número de fios de execução verificar-se-á uma degradação de *performance* nestas situações, pois os vetores de acumulação são relativamente grandes, e cada *thread* deverá iterar sobre o mesmo para o cálculo do respetivo centroide.

IV. Resultados

Os resultados foram obtidos correndo o executável numa máquina com o seguinte *hardware*:

- GeForce RTX 3050 Mobile (Arquitetura Ampere, Tamanho cache L1:
  - Arquitetura Ampere
  - Tamanho Cache L1: 128KB/SM
  - Tamanho Cache L2: 2MB
  - Tamanho memória: 4GB
  - Versão CUDA: 11.8
- 11th Gen Intel i5-11400H (12) @ 4.500GHz
- 16GB Memória RAM

O executável foi compilado com *nvcc -O3 -DNDEBUG*, e posteriormente corrido com a ferramenta *hyperfine*, num total de 5 execuções para cada combinação de valores de *input*, sem aquecimento prévio. Daí, apresenta-se a média deste *benchmarking* para a análise de resultados que se segue. O critério de paragem usado foi o número de iterações, fixado nas 20, tal como indicado no enunciado anterior. Os valores de *input* para o número de amostras variou entre 15 mil, 240 mil e 10 milhões, sendo estes primeiros obtidos de acordo com os tamanhos das *caches* apresentados em cima e tendo em consideração que cada amostra ocupa 8 *bytes* em memória (correspondentes a dois valores de vírgula flutuante de precisão simples). O número de *clusters* varia entre 4, 20 e 32. Mencionar, ainda, que o valor total de *threads* em cada teste é da seguinte forma: $threads = 32 \times n, \ n \in [0, 10] \cap \mathbb{N}$. Para obter tais valores, testou-se tanto o caso em que se fixava o número de blocos em 32 e se variava o número de fios de execução, como o oposto. Distinguem-se nos gráficos em seguida pelos sufixos *FB (fixed blocks)* ou *FT (fixed threads)*, respetivamente.

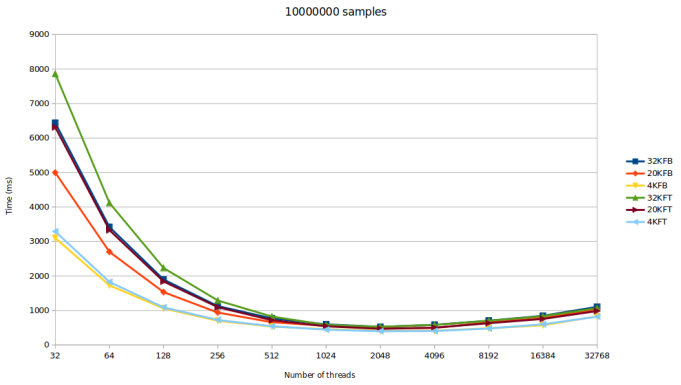

Figura 1. Tempos de execução para 10 milhões de amostras

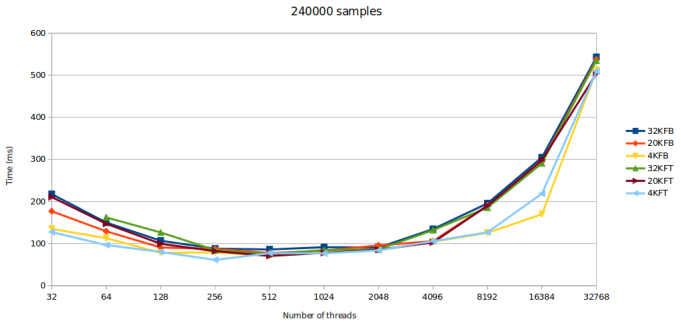

Figura 2. Tempos de execução para 240 mil amostras

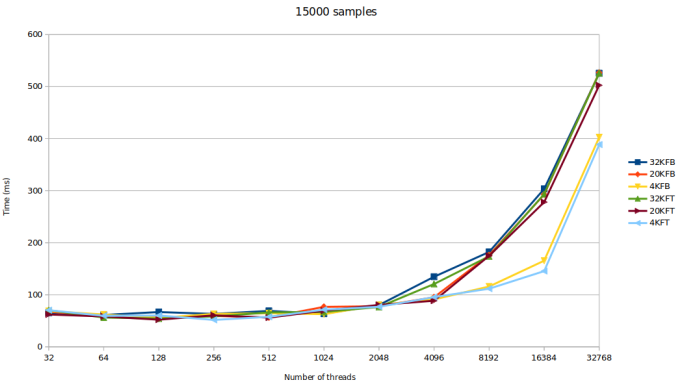

Figura 3. Tempos de execução para 15 mil amostras

V. Análise de resultados

No primeiro caso, com 10 milhões de amostras, é notório que o valor de ideal de *threads* se encontra por volta de 1024, com tempos de execução muito semelhantes em qualquer caso. O *overhead* do aumento do número de *threads* também evolui de forma semelhante. Isto deve-se ao facto de que, numa perspetiva mais ampla, a variação do número de *clusters* não é tão significativa quanto isso, considerando que o número de amostras é várias ordens de magnitude superior.

No entanto, há que destacar a forma como o programa se comporta nos primeiros testes, ou seja, com 32 *threads*. Contrariamente ao que seria esperado, lançar 32 *threads* num único bloco é **menos** eficiente do que o oposto. Em termos de *hardware*, as *threads* de um mesmo bloco são escalonadas em grupos de 32, conhecidos por ***warps***. No caso de serem lançadas um valor N inferior a 32, teremos a $32 - N$ *threads "masked out"*, o que teoricamente resultaria em degradação de *performance* devido à subutilização dos recursos.

Nos outros casos, o valor ideal do número de fios de execução também tende a situar-se entre os 512 e os 1024, sendo que nestas regiões de valores os para os números de *clusters* testados não aparentam ter impacto, à semelhança do que acontece para os 10 milhões de amostras, por razões semelhantes.

Por último, nestes dois últimos gráficos consegue-se visualizar a situação antevista na secção III, em que com aumentando o número de *threads* mas mantendo o número de *clusters* a *performance* é impactada negativamente, com um crescimento acentuado do tempo de execução.

VI. Conclusão

Em jeito de conclusão, é importante destacar as dificuldades sentidas pelo grupo. Em concreto, o paradigma de programação em GPU é substancialmente diferente e acarreta os seus próprios desafios, nomeadamente as questões das transações de memória entre *host* e *device*, a necessidade de reduzir (ou mesmo eliminar) sincronização/exclusão mútua e como efetuar os acessos à memória do próprio dispositivo (*coalescing*).

Por outro lado, é relevante fazer menção ao facto de não termos conseguido obter *profiling* da aplicação. No contexto de computação paralela, esse aspeto é fundamental para estudar métricas como acessos à memória, pelo que seria algo a explorar futuramente.

Referências

[1] "CUDA C++ Programming Guide" (2022, December) https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf