Trabalho Prático 3

Computação paralela

1st Francisco Reis Izquierdo
Universidade do Minho
Mestrado em Engenharia Informática
Aveiro, Portugal
pg50384@alunos.uminho.pt

1st Francisco Alberto do Fundo Novo
Universidade do Minho
Mestrado em Engenharia Informática
Braga, Portugal
pg50374@alunos.uminho.pt

***Abstract*—O presente relatório tem como objetivo ilustrar todo o trabalho feito pelo grupo relativo à última fase no qual, ao longo do mesmo, estarão detalhadas toda a implementação bem como técnicas de paralelismo em programas e correspondente justificação.**

***Index Terms*—CUDA, paralelismo, tempos de execução, resultados.**

I. Introdução

No âmbito da unidade curricular de Computação Paralela foi desenvolvido o último trabalho prático no qual o foco passou por retificar e alargar os conceitos abordados na anterior fase, com o intuito de melhorar a performance ao analisar vários aspetos no algoritmo desenvolvido através de vários conceitos de paralelismo de programas com recurso à ferramenta **CUDA** abordada nas aulas práticas da UC.

II. Alterações

Dado o objetivo de melhorar e extender a solução paralelizada da fase anterior, primeiramente focou-se em analisar quais as regiões do código que podiam ser corrigidas e que eram bloqueadores de performance. Desta forma, conseguimos perceber que havia duas regiões que deveriam ser retificadas:

- Região sequencial: na qual o foco passou por paralelizar esta região, tendo o devido cuidado para não haver *data races*. Fazem parte desta região, as funções:
  - *recalculateCentroids*: responsável por recalcular os novos centroides;
  - *clean*: responsável por resetar a informação de cada cluster, para uma nova iteração;
- Região paralela: na qual a atenção passou por extender e melhor conceptualizar a implementação previamente feita, fazendo recurso a conceitos alargados de paralelismo.

Posto isto, o âmbito passou por paralelizar o algoritmo *kmeans* de forma completa, onde se abandonou os conceitos implementados em **OpenMP** na fase anterior para adotar o ambiente **CUDA**. Os motivos para tal abordagem foram o facto desta plataforma permitir aproveitar o enorme paralelismo trazido pela *GPU* no que concerne ao processamento de números, bem como a experiência de trabalho com novas ferramentas.

III. Implementação

Uma vez usado o ambiente **CUDA**, dois fatores que ajudaram na implementação desta fase foram os conceitos relativos ao número de blocos usados e o número de *threads* por bloco. Além disso, a implementação foi dividida em várias componentes, dando ênfase aos seguintes tópicos:

- Alteração das estruturas de dados;
- Eliminação das regiões sequenciais;
- Invocação dos *kernels* com recurso ao **CUDA**;

A. *Alteração das estruturas de dados*

Por forma a melhor organizarmos e tornarmos mais legível o código, bem como fazer um uso apropriado dos princípios de localidade foram efetuadas algumas alterações em relação às estruturas de dados. Assim, foram removidos os arrays respetivos aos valores das componentes *x* e *y* para cada ponto/centroide e passamos a ter as seguintes estruturas de dados:

- ***Point***: Estrutura de dados que representa um ponto de duas dimensões, *x* e *y;*
- ***ClusterInfoParcial***: Estrutura de dados que contém a informação relativa às somas parciais das componentes *x* e *y* de cada cada cluster, bem como a quantidade parcial de elementos.

B. *Eliminação das regiões sequenciais*

Dado a abordagem ao **CUDA**, teremos agora de ter em mente o uso apropriado do número de blocos e o número de *threads* por bloco, de forma a mitigarmos tanto as regiões sequenciais, bem como evitar *data races*. Deste modo, a lógica utilizada passa por inicialmente serem enviados para a **GPU** as seguintes estruturas de dados:

- ***sample***: Array de pontos pertences à amostra;
- ***centroids***: Array que contém os centroides de cada *cluster*;
- ***elemensTotal***: Array que contém para cada *cluster* contém quantos elementos tem;
- ***infos***: Array que contém as informações parciais de cada *cluster*, como o somatório parcial das componentes *x*, *y* e quantidade parcial dos pontos que pertencem a um determinado *cluster*;

Seguidamente, o algoritmo *kmeans* corre de forma totalmente paralelizada no *GPU*, devido à implementação de dois *kernels*.

- ***kmeansKernel***: *Kernel* responsável por calcular a que *cluster* um dado ponto pertence;
- ***recalculateCentroidsKernel***: *Kernel* responsável por atualizar os centroides de cada *cluster*, bem como resetar as informações temporárias, nomeadamente, os somatórios parcias supramencionados;

C. *Kernel kmeansKernel*

Na invocação deste *kernel*, o número de blocos e *threads* por bloco são passados como parâmetros na invocação do programa, cabendo ao programador fazer uma balanceamento entre a quantidade de *threads* usar dado o tamanho da amostra.

- Cada *thread* trata de um conjunto de pontos, no qual para cada um calcula a que *cluster* pertence;
- À medida que cada *thread* verifica a que *cluster* um dado ponto pertence, realiza para o *cluster* correspondente o somatório parcial da componente *x*, componente *y* e incrementa o número parcial de elementos;

Por forma a facilitar a lógica de raciocínio, temos então um espaço representado por uma matriz de tamanho **T** por **K** (com **T** o número total de *threads* ativas e **K** o número total de *clusters*), em que as colunas dizem respeito a cada *thread* e as linhas a cada *cluster*. Desta forma, cada *thread* apenas acede à sua coluna respetiva, no qual cada elemento da matriz diz respeito à estrutura de dados relativa a cada *cluster* que contém a informação relativa de cada um. Deste modo, em cada iteração é realizado o pré-processamento de cada *cluster* sem ocorrerem *data races*.

$$\begin{bmatrix} thread\mathbf{1}Cluster\mathbf{1} & ... & thread\mathbf{T}Cluster\mathbf{1} \\ thread\mathbf{1}Cluster\mathbf{2} & ... & thread\mathbf{T}Cluster\mathbf{2} \\ ... & ... & ... \\ thread\mathbf{1}Cluster\mathbf{K\text{-}1} & ... & thread\mathbf{T}Cluster\mathbf{K\text{-}1} \\ thread\mathbf{1}Cluster\mathbf{K} & ... & thread\mathbf{T}Cluster\mathbf{K} \end{bmatrix}$$

D. *Kernel recalculateCentroidsKernel*

Na invocação deste *kernel*, o número de blocos é constante, sendo este 1 e o número de *threads* por bloco é igual ao número de *clusters*, onde teremos para cada *cluster* uma *thread* a atualizar a informação deste ao analisar a informação contida na matriz supramencionada, ao realizar o somatório das componentes parciais previamente realizadas. Assim, para cada *cluster*, a informação parcial encontra-se distribuída por linha, sendo que cada *thread* irá aceder a endereços de memória contíguos, tirando proveito da **localidade espacial**. Além disto, cada *thread* ao atualizar a informação reseta os valores de cada elemento da matriz para a próxima iteração. Por fim, cada *thread* ao ter o somatório das componentes *x* e *y*, bem como o número de elementos de cada *cluster*, calcula o novo centroide.

IV. CPU e GPU

Além de toda a explicação previamente detalhada, convém salientar o papel e as diferenças no que diz respeito ao uso do **CPU** e ao uso do **GPU**, uma vez que ambos são utilizados em prol de objetivos diferentes.

- **CPU**: Utilizado para alocar memória nas estruturas de dados, bem como gerar a informação inicial de cada uma dessas estruturas, como pontos da amostra e informação inicial de cada *cluster*. Responsável por alocar espaço suficiente e copiar para o **GPU** todos os dados relativos às estruturas de dados. Realiza as iterações e correspondentes chamadas pelos *kernels*.
- **GPU**: Utilizado para computar toda a informação proveniente e previamente alocada pelo **CPU**. É onde o cerne do processamento do algoritmo decorre, de forma paralelizada e assegurando a não existência de *data races*. É invocado duas vezes, dado que temos dois *kernels* a cada iteração do algoritmo.

V. Ambiente de execução

Todos os testes de performance realizados tiveram o mesmo ambiente de execução sendo disponibilizado pela infraestrutura de computação *SeARCH*, abordada nas aulas práticas da UC e que teve um papel fulcral no apoio ao desenvolvimento das várias fases do projeto. Além disso, convém salientar que de entre as várias partições apresentadas pelo *SeARCH*, o ambiente de execução ficou restringido à partição usada nas aulas práticas da disciplina, isto é, à partição ***cpar*** que se insere no segmento ***Ivy Bridge*** pertencente à *Intel*. Posto isto, de entre as várias especificações da partição supramencionda no qual foram realizados os diversos testes, temos a seguinte tabela de especificações:

CPU	***clocks (GHz)***	Cores	Memória (GB)	Acelerador
E5-2670v2	2.50	20	64	Tesla K20m

Além disto, convém realçar o acelerador **Tesla K20m** dado que é o **GPU** usado para computacionar todo o algoritmo desenvolvido. Assim, de entre várias especificações, destacam-se as seguinte:

- Hierarquia de memória (detalhado mais à frente);
- Dimensão máxima de uma grelha de blocos de *threads*: 3;
- Tamanho do *warp*: 32;
- Quantidade de *threads* máximas por bloco: 1024;

VI. Previsão do perfil de execução

Uma vez detalhado o processo lógico por detrás da implementação, bem como as especificações do ambiente de execução, podemos agora tentar prever com base na informação demonstrada, o comportamento do programa face a várias vertentes:

- Número total de *threads* invocadas (número de blocos e número de *threads* por bloco);
- Número de *clusters*;
- Dimensão da amostra;

No entanto, existem outras nuances a ter em conta, no que concerne à operabilidade dos compenetes:

- Copiar dados: operações de copiar dados do **GPU** para a memória, teoricamente são mais custosas;
- Princípio de localidade: o **GPU** também usufrui das características relativas ao princípio de localidade, podendo ser otimizado nas vertentes de localidade espacial e temporal;
- Limite de capacidade: o **GPU** tem um limite de capacidade no que concerne à quantidade de *threads* que podem ser lançadas, bem como da memória a usar;

Posto isto, podemos averiguar o funcionamento do algoritmo fase às várias vertentes descritas:

- Irá haver um equilíbrio ideal no que diz respeito ao número de *threads* a usar em prol da dimensão da amostra e também do número de *clusters*;
- Um grande número de *threads* não será sinónimo de melhor performance;
- Iremos ver no perfil de execução que o algoritmo passa mais tempo no **GPU** do que no **CPU**;
- Irá haver funções que são computacionalemnte mais custosas que outras;

Dadas as supramencionadas previsões, de forma a compará-las com a realidade foram realizados diversos testes que extendem-se e abrangessem os vários tópicos a analisar.

VII. Testes e Escalabilidade

De forma a averiguar a magnitude dos vários parâmetros previamente referidos foram realizados os devidos testes e a correspondente análise teórica da performance do programa face ao intervalo de valores relativos aos parâmetros, no qual temos as seguintes avaliações:

- Teste 1: Tamanho da amostra fixo, variando o número de *threads* totais para o número de *cluster* igual a 4 e 32;
- Teste 2: Tamanho da amostra variado, variando o número de *threads* totais para o número de *clusters* igual a 4 e 32;
- Teste 3: Tamanho da amostra variado, com o número de *threads* totais fixas, sendo este número a "melhor média" relativamente a testes anteriores;

A. Teste 1

Neste teste, tal como descrito em cima, foram realizados vários testes com um tamanho da amostra fixo onde **N** = 10000000 de pontos, fazendo variar o número total de *threads* e para dois números de *clusters*, com **K** = 4 e **K** = 32. Além disto, importa salientar que para cada bloco de *threads*, de forma a tirar partido da capacidade do mesmo, maximizou-se o número total de *threads* a usar com atenção no limite disponível de **1024** *threads* por bloco.

Threads	Blocos	Tempo (seg)
128	1	5.37
512	1	1.91
2048	2	1.03
5120	5	0.57
8192	8	0.60
16384	16	0.92
32768	32	1.46

TABLE I: Nº de *clusters* = 4, Dimensão = 10000000

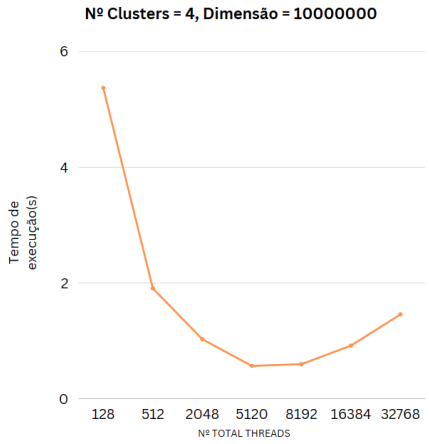


Fig. 1: Visualização gráfica da tabela 1

Threads	Blocos	Tempo (seg)
128	1	17.79
512	1	6.00
2048	2	2.62
5120	5	1.38
8192	8	1.16
16384	16	1.52
32768	32	3.06

TABLE II: Nº de *clusters* = 32, Dimensão = 10000000

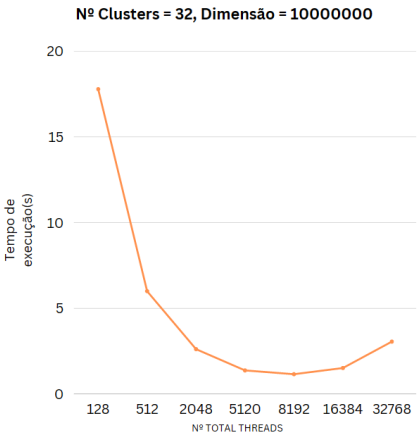


Fig. 2: Visualização gráfica da tabela 2

Dadas as métricas e valores obtidos através da realização de testes nesta vertente, conseguimos perceber que existe para uma amostra com uma dada dimensão um limite superior e inferior imposto no que concerne ao número de *threads* a usar. Assim, conseguimos perceber que devemos aumentar o número de *threads* com o intuito de melhorar a performance e reduzir o tempo de execução, mas sem impor um *overhead* com excesso de *threads*. Este *overhead* imposto pelo aumento do número de *threads* deve-se ao facto da estrutura de dados, que alberga as informações parciais de cada *cluster*, ter um tamanho igual a **K** vezes o número total de *threads*, sendo este posteriormente enviado para o **GPU**, o que implica que quanto maior o número de *threads* maior o tamanho da estrutura de dados.

B. Teste 2

No que concerne ao teste 2, este foi realizado tendo em conta as especificações da hierarquia de memória relativas ao ambiente de execução **Tesla K20m**, nomeadamente averiguando tamanhos variados da amostra em prol das mesmas. Assim, das especificações previamente definidas temos:

- Cache nível 1 (L1): A cache nível 1 possui até 64Kib de memória (65536 *bytes)*;
- Cache nível 2 (L2): A cache nível 2 possui até 256Kib de memória (262144 *bytes*);
- Memória principal: A memória principal do *GPU* é de 5GB;

Assim, foram realizados os testes com tamanhos de amostras variados, com o intuito de analisar como a dimensão da amostra pode afetar a performance de execução relativamente às especificações e tamanhos dos componentes de memória. Para tal, foram efetuados alguns cálculos com base na estrutura de dados que representa um ponto. Esta, tal anteriormente referido, é constituida por dois valores, associados às componentes x e y do ponto respetivo, sendo do tipo *float*. Assim, temos os seguintes tópicos:

- Cada ponto ocupa 8 *bytes*;
- Se dividirmos o tamanho de um dado nível de memória, conseguimos perceber quantos pontos cabem nesse mesmo nível de memória;
- Na cache de nível 1, cabem então até 8192 pontos;
- Na cache de nível 2, podem estar então até 32768 pontos;
- Para a memória total, cabem então até 625000000 de pontos;

Posto isto, foram realizadas várias dimensões de amostras, com tamanho inferior e superior a cada nível de memória. Com isto, temos amostras de tamanho:

- 1000 pontos: Amostra com dimensão menor que o tamanho da cache de nível 1;
- 10000 pontos: Amostra com dimensão maior que o tamanho da cache nível 1 e dimensão menor que o tamanho da cache nível 2;
- 100000 pontos: Amostra com dimensão maior que o tamanho da cache nível 2 e dimensão menor que o tamanho da memória total;
- 1000000 pontos: Amostra com dimensão maior que o tamanho da cache nível 2 e dimensão menor que o tamanho da memória total;

Com base nisto, os diversos testes foram repartidos por dois números de *clusters*, K=4 e K=32, a fim de explorar como a quantidade de *clusters* afeta a performance, mas também o número de *threads* totais.

1) 4 clusters: Para 4 *clusters* e um número crescente de *threads*, para cada dimensão da amostra previamente definido, temos os seguintes resultados.

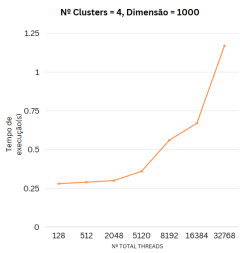


Fig. 3: Variação do tempo execução em relação ao número de threads para K=4 e N=1000

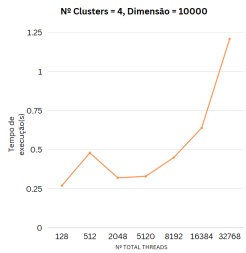


Fig. 4: Variação do tempo de execução em relação ao número de threads para K=4 e N=10000

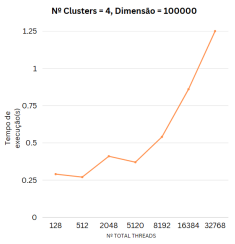


Fig. 5: Variação do tempo de execução em relação ao número de threads para K=4 e N=100000

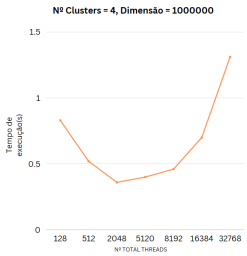


Fig. 6: Variação do tempo de execução em relação ao número de threads para K=4 e N=1000000

2) 32 clusters: Para 32 *clusters* e um número crescente de *threads*, para cada dimensão da amostra previamente definido, temos os seguintes resultados.

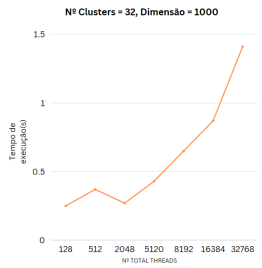


Fig. 7: Variação do tempo de execução em relação ao número de threads para K=32 e N=1000

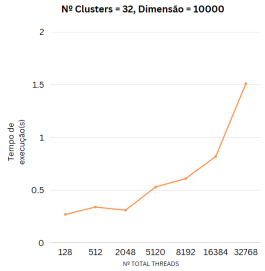


Fig. 8: Variação do tempo de execução em relação ao número de threads para K=32 e N=10000

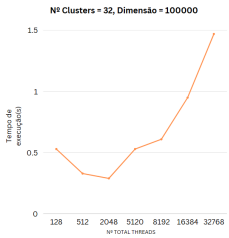


Fig. 9: Variação do tempo de execução em relação ao número de threads para K=32 e N=100000

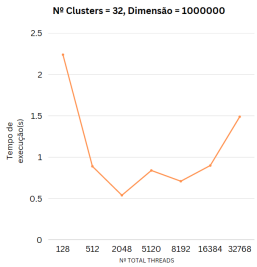


Fig. 10: Variação do tempo de execução em relação ao número de threads para K=32 e N=1000000

Podemos concluir que a dimensão da amostra tem impacto direto na performance, dado que quanto maior for a dimensão da amostra, consoante o número de *threads*, tem um tempo de execução maior. Além disso, conseguimos perceber um padrão sobre o número de *threads* a usar com as seguintes propriedades:

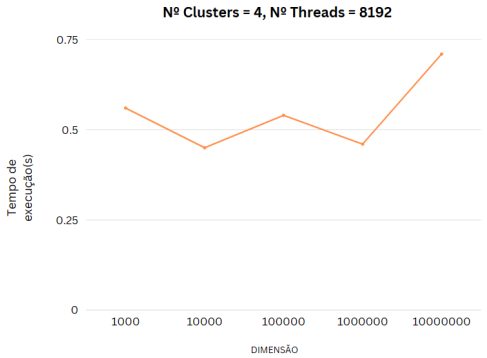
- Para dimensões pequenas, um número consideravelmente pequeno de *threads* satisfaz uma boa performance e à medida que vamos aumentando a quantidade de *threads* utilizadas percebemos que a performance se degrada;
- À medidade que aumentamos a dimensão da amostra, percebemos que um número de *threads* pequeno não traz bons resultados de execução;
- À medida que aumentamos a quantidade de *threads* com o aumento da dimensão, percebemos que a performance vai melhorando até um certo ponto, tal como referido no teste 1;
- Apesar de haver um aumento da dimensão da amostra, aumentar significativamente o número de *threads* implica uma degradação da performance;
- O número de *threads* a utilizar depende efetivamente da dimensão da amostra, havendo um balanço entre ambos;

C. Teste 3

Para o teste 3, foram realizados testes com dimensões variadas da amostra e um número constante de *threads*, para averiguar se entre duas amostras de dimensões diferentes,(neste caso 10 vezes maior), existe alguma relação bem como se o melhor número de *threads* para uma dada amostra é o melhor número de *threads* de outra amostra.

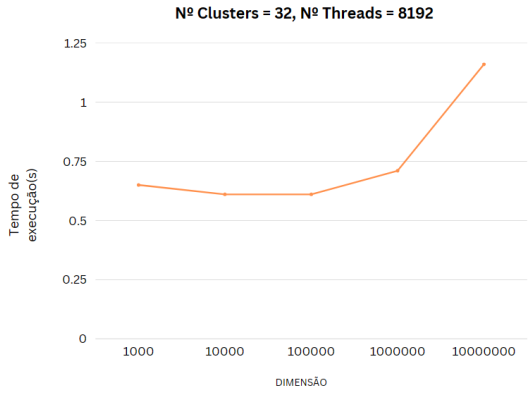
Size	Threads	Blocos	Tempo(seg)
1000	8192	5	0.56
10000	8192	5	0.45
100000	8192	5	0.54
1000000	8192	5	0.46
10000000	8192	5	0.60

TABLE III: Nº de *clusters* = 4 e Nº de *threads* fixo



Size	Threads	Blocos	Tempo(seg)
1000	8192	5	0.65
10000	8192	5	0.61
100000	8192	5	0.61
1000000	8192	5	0.71
10000000	8192	5	1.16

TABLE IV: Nº de *clusters* = 32 e Nº de *threads* fixo



O mesmo número de *threads* para amostras de diferentes dimensões pode trazer um aumento como uma degradação da performance. Além disso, existe uma correlação entre a quantidade de *clusters* utilizados para um mesmo número de *threads*, no qual um aumento do número de *clusters* para amostras de dimensões crescentes e para o mesmo número de *threads*, o tempo de execução aumenta.

D. Repartição do tempo

Tal como fora anteriormente referido, foi feita a previsão de como seria repartido o tempo de execução entre os componentes, no qual após os diversos testes, foram efetuadas médias de tempos de execução relativos à partição do tempo, no qual resultou o seguinte gráfico.

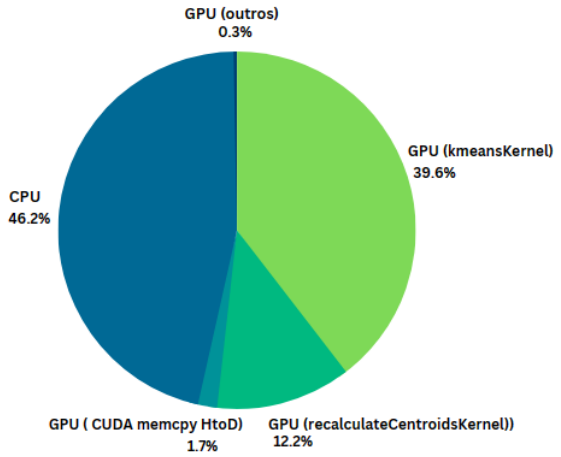


Fig. 11: Repartição dos tempos de execução

Podemos observar que contrariamente ao esperado, existe uma partição quase igual entre o tempo dispendido entre o **CPU** e o **GPU**. Isto deve-se ao facto de que, apesar do algoritmo *kmeans* ser inteiramente computado no **GPU**, cerca de metade do tempo é dispendido no **CPU** devido à criação das amostras, à alocação de memória, atribuição dos dados às estruturas correspondentes e envio dos dados para o **GPU**. Além disto, conseguimos perceber que grande parte do tempo gasto pelo **GPU** é focado no *kernel kmeansKernel* que efetua o cálculo da atribuição de cada ponto ao respetivo *cluster*, sendo o restante tempo relativo ao *kernel recalculateCentroidsKernel*.

VIII. Análise de escalabilidade

No que concerne à escalabilidade, convém salientar duas vertentes.

A. Comparação

Iremos então comparar as duas melhores versões relativamente a um mesmo número de *clusters* e a uma mesma dimensão da amostra, respetivamente 4 e 32 *clusters* e 10000000 de pontos. A implementação em **CUDA** permitiu uma melhor extensão e implementação de métricas de paralelismo, no qual na versão atual, o algoritmo *kmeans* é completamente paralelizado, contrariamente ao que acontecia à versão anteriormente implementada, no qual havia duas componentes para o algoritmo *kmeans*, sendo uma paralela e outra sequencial. Assim, a parte do algoritmo sequencial foi motivadora, quando comparado com a versão atual, dum pior desempenho.

Versão	Clusters	Dimensão	Tempo (s)
OpenMP	4	10000000	0.89
CUDA	4	10000000	0.57
OpenMP	32	10000000	1.54
CUDA	32	10000000	1.16

TABLE V: Versão anterior vs Versão atual

B. Escalabilidade da solução desenvolvida

No que concerne à solução desenvolvida, ao fazermos um âmbito geral da escalabilidade, conseguimos perceber que consoante o número de *threads* a usar deve ter em conta a dimensão da amostra bem como a quantidade de *clusters*. Além disto, para dimensões relativamente pequenas, devemos usar um número pequeno também de *threads* de forma a não causar uma imposição desnecessária de demasiadas *threads*. Com amostras de dimensão maior devemos escolher o número de *threads* apropriado, tendo sempre em conta a não exagerar neste número dado o *overhead* visualizado nos cenários de teste. Convém sublinhar que para dimensões maiores que as testadas conseguimos prever que a solução implementada está apta e confere escalabilidade tendo sempre em atenção ao número de *threads* a usar. Conseguimos perceber que apesar de não haver uma medida inteiramente correta para prever que quantidade de *threads* usar podemos utilizar a seguinte métrica:

- O número de *threads* a utilizar, quando especificado o número de blocos a usar, é igual à raíz quadrada do produto da dimensão da amostra com o número de blocos (este valor apenas tem impacto no *kernel kmeansKernel*);

Salienta-se que a especificação do número de *threads* é fulcral para o desempenho do *kernel kmeansKernel*, uma vez que é computacionalemente mais despendioso e o *kernel recalculateCentroidsKernel* tem um número de *threads* igual à quantidade de *clusters* em questão, logo o foco de definir um correto número de *threads* irá afetar o desempenho do *kernel* supramencionado. Também podemos dar ênfase à dimensão da amostra comparativamente aos níveis de memória, uma vez que dimensões de amostras que consigam ser inseridas totalmente no nível de hierarquia adequado, terão as leitura e escritas efetuadas mais rapidamente, no qual levará a melhores perfomances.

IX. Conclusão

A realização deste projeto permitiu consolidar noções acerca de correção e eficiência de programas paralelos com o auxílio da plataforma **CUDA**, tornando claro que as escolhas tomadas na implementação determinaram o desempenho do programa.