# Work Assignment - Phase 3

Parallel Computing, Master's Degree in Computer Engineering, University of Minho

**Afonso Miguel Matos Bessa**
MEI
University of Minho
pg53597

**Francisco Luís Rodrigues Claudino**
MEI
University of Minho
pg50380

***Abstract*—Este documento representa a última etapa na evolução do código de simulação de dinâmica molecular aplicado aos átomos de gás árgon. Ao longo deste relatório, serão examinadas potenciais otimizações adicionais, com recurso a aceleradores como o CUDA, acompanhadas das suas implementações e testes. Os resultados serão ainda comparados com os dois trabalhos anteriores, proporcionando uma avaliação abrangente do projeto.**

## Introdução

Na última parte deste projeto, a nossa equipa desenvolveu uma solução onde a carga de trabalho intensiva, ou seja, *hot-spots*, fosse distribuída entre threads numa CPU, reduzindo o tempo total de execução. Para melhorar ainda mais o tempo de execução do código, a nossa equipa utilizou técnicas de paralelismo eficientes, seja aumentando a quantidade de *threads* disponíveis sem reduzir a sobrecarga, ou dividindo as tarefas entre vários processos.

Para esta última abordagem, o uso de uma **MPI** (*Message Passing Interface*) é essencial, pois permite a comunicação e transferência de dados entre processos. Com o uso de muitos processos independentes, cada um fica atribuído a uma tarefa específica dentro do programa, tornando possível a criação uma *pipeline* dentro de uma aplicação, reduzindo o tempo de execução.

Quanto à execução de muitas *threads*, é amplamente documentado que as GPUs são otimizadas e desenvolvidas em torno do processamento de grandes blocos de dados não segmentados, apresentando uma afinidade extrema com o paralelismo, onde um maior número de *threads* é utilizado para processar esses blocos de dados simultaneamente, com pouco ou nenhum *overhead*. Para aceder a esses benefícios, uma API como o **CUDA** (*Compute Unified Device Architecture*) ajudará a nossa equipa a desenvolver o programa de forma mais eficiente.

## Revisão do Trabalho Efetuado

Sendo esta a última fase deste projeto, iremos efetuar uma breve menção às fases transatas, mencionando desta forma as principais alterações efetuadas até este ponto.

### *Versão Sequencial do Código*

Nesta fase do trabalho prático, começamos entender o código e por efetuar o `code profiling`, analisando as funções que consomem maior tempo de execução.

Os *bottlenecks* em termos de tempo de execução estariam principalmente nas funções *Potential* e *computeAccelerations*, como podemos observar na Figura 1.

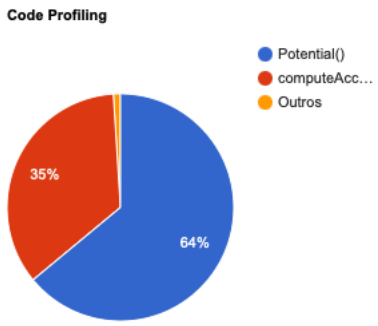


Fig. 1: Code Profiling - Fase 1

Após uma análise mais detalhada do código em geral e das duas funções mencionadas anteriormente em particular, efetuamos as seguintes alterações ao código:

1) Remoção das funções *pow*, devido ao *overhead* causado pela sua invocação, primeiramente devido a mudança do *frame-pointer* e devido também ao facto de que o compilador de C não otimiza funções de bibliotecas externas.
2) Otimização da função *Potential* - dado que a complexidade desta função é quadrática ($O(n^2)$) e que o ciclo que percorre o vetor das acelerações possui uma redução, torna-se impossível a vetorização desta função. Dado isto, alteramos o ciclo aninhado para começar em $j = i+1$ em vez de 0, para evitar operações redundantes e também alteramos o cálculo da energia potencial. A energia potencial inicialmente era calculada da seguinte forma:

   ```
   Pot += 4 * epsilon * (term1 - term2)
   ```

   e passou a ser calculada da seguinte forma:

   ```
   Pot += (1-rSqd3)/rSqd6
   ```

   onde *rSqd3* é a distância entre as partículas ao quadrado elevado a 3 e *rSqd6* é esta mesma distância elevada a 6. A variável f era calculada através da seguinte fórmula:

   Por fim, efetuamos *loop unroll* do ciclo que efetuava o cálculo da distância entre as partículas para aumentar o *Instruction Level Paralelism*.
3) Já na função *computeAccelerations* alteramos o cálculo da variável f que era calculada através da seguinte fórmula:

   ```
   f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4))
   ```

   e passou a ser calculada da seguinte forma:

   ```
   f = ((48-24*rSqd3)/(rSqd6*rSqd))
   ```

   sendo as varíavies as mesmas das mencionadas acima.
4) Efetuamos também a vetorização das outras funções, que mesmo não consumindo uma fração considerável de tempo melhora a

hierárquia de memória e diminui o número de *cache misses*, bem como, o número de acessos a memória.

A última otimização efetuada foi a junção das duas funções mencionadas acima devido ao comportamento similar que apresentam, bem como, ao uso e cálculo das mesmas variáveis (distancia entre as partículas).

Com estas otimizações efetuadas, conseguimos diminuir drasticamente o tempo de execução, passando de 236 segundos para 4.7 segundos, e também diminuímos o número de ciclos, o número de instruções e a percentagem de *cache misses*.

### *OpenMP*

Na segunda fase do trabalho prático, foi necessário efetuar novamente o *code profiling* para podermos identificar *hot-spots* no nosso código de forma a utilizar as diretivas de **OpenMP** apenas nas funções que ocupassem maior tempo de execução, dado que o *overhead* do uso destas diretivas é considerável e, por isto, só devem ser usadas em funções que possam constituir um *bottleneck* no programa.

Após efetuado o *code profiling* podemos observar através da Figura 2 que a função *computeAccelerationsPotential* ocupa quase todo o tempo de execução do programa, logo o foco do uso das diretivas OpenMP ficou nesta função.

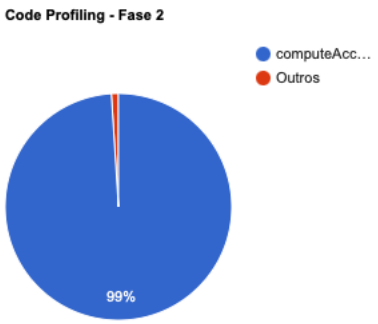


Fig. 2: Code Profiling - Fase 2

Após uma análise detalhada da função `ComputeAccelarionsPotencial`, foi possível identificar três momentos onde paralelizar o código poderia ser benéfico, sendo estes os seguintes:

1) O ciclo que inicializa o *array* das acelerações a zero;
2) O ciclo que calcula a energia potencial de cada partícula;
3) O ciclo aninhado no ciclo anterior que calcula a força necessário para o cálculo da energia potencial.

Como os três blocos de código analisados consistiam em ciclos, a diretiva `#pragma omp parallel for` foi implementada pois sem ela, não seria possível utilizar fios de execução nestes ciclos. Após uma análise cuidadosa, optamos por utilizar as seguintes diretivas:

1) **#pragma omp simd** - Foi decidido utilizar pois consideramos fundamental a vetorização de instruções dentro de ambos os ciclos, sendo que o *overhead* que esta diretiva implica é compensado pela *performance* que os fios de execução oferecem.
2) **#pragma omp reduction** - Foi utilizado de modo a impedir as *data races* no acesso ao *array* global das acelerações e com o objetivo de reduzir as operações matemáticas de duas variáveis, devido aos cálculos de valores consideravelmente elevado e visto que o *overhead* é relativamente pequeno quando comparado ao tempo de execução ganho em cada iteração.
3) **#pragma omp scheduled (dynamic, 50)** - Optamos por usar esta diretiva em detrimento das outras três (*guided, static, runtime*) pois é aquela que balanceia de forma mais uniforme a carga computacional de cada fio de execução.

Com estas otimizações efetuadas, conseguimos reduzir o tempo de execução, sendo feita uma análise de escalabilidade que será abordada com mais detalhe aquando da comparação das tres versões de código implementadas, na discussão dos resultados.

## MPI - Message Passsing Interface

O **MPI** é um padrão de programação que é utilizado em computação paralela, tendo como ponto forte a divisão de tarefas de um algoritmo entre vários processos que, ao processarem os dados em ”blocos”, permite a criação de uma *pipeline* eficiente.

De referir também que o **MPI** faz uso de uma arquitetura com memória distribuída, ou seja, cada processo possui a sua própria memória privada e, consequentemente, as suas próprias variáveis, eliminando assim quaisquer *data races* que possam existir.

Dadas estas considerações e tendo em mente a análise introdutória feita no capítulo anterior, decidimos não implementar qualquer versão do programa que utilizasse **MPI**, dado que para a secção de código que beneficiaria mais do uso de **MPI**, que seria a função *computeAccelerationsPotential()*, necessita que o vetor das acelerações fosse acedido por todos os processos em simultâneo, para que desta forma o tempo de execução desta função específica diminua (que constitui o grande e único *hot-spot* do programa).

Assim, como o **MPI** possui uma arquitetura de memória distribuída, este vetor não seria partilhado por todos os processos, pois todos possuiriam uma cópia deste vetor na sua memória partilhada, tendo de haver comunicação entre eles para todos conseguirem efetuar as operações pretendidas nas diferentes posições do vetor (divisão da carga de trabalho de forma eficiente), o que resultaria num número relativamente grande de troca de mensagens e num uso excessivo e desnecessário de memória que diminuiria a performance do programa elevando o tempo de execução do mesmo.

Concluindo, não consideramos que a utilização do **MPI** seria benéfica para o nosso programa, tanto em termos de *performance* como em termos de escalabilidade, logo não foi utilizado. Poderia ser implementada uma versão que tirasse proveito deste padrão de programação, mas tendo em conta as considerações feitas anteriormente, não foi desenvolvida.

## CUDA - Compute Unified Device Architecture

Conforme mencionado anteriormente, esta plataforma de computação paralela possibilita tirar partido de determinados tipos de unidades de processamento gráfico (GPUs) para realizar o processamento. As GPUs revelam-se mais eficazes do que as unidades de processamento central (CPUs) em situações em que o processamento de extensos blocos de dados ocorre em simultâneo, como é o caso da função `computeAccelerationsPotential()`. O **CUDA** disponibiliza uma região de memória partilhada de elevada velocidade, que pode ser partilhada entre *threads*. Este recurso pode ser utilizado como uma *cache* gerida pelo utilizador, proporcionando uma maior largura de banda.

O **CUDA** permite a utilização de dois tipos diferentes de memória: **global** e **partilhada**, sendo que a Memória Partilhada é ligeiramente mais rápida do que a Global. Optamos por utilizar a memória partilhada para as operações de adição, uma vez que, devido à necessidade de utilizar adição atómica para evitar possíveis conflitos de dados

(*Data Races*), as operações de adição tornar-se-iam mais lentas. Assim, esperávamos acelerar este processo ao utilizar memória partilhada.

A Figura 3 ilustra a metodologia que o grupo seguiu de maneira a implementar a versão do programa com o auxilio da plataforma de computação paralela **CUDA**.

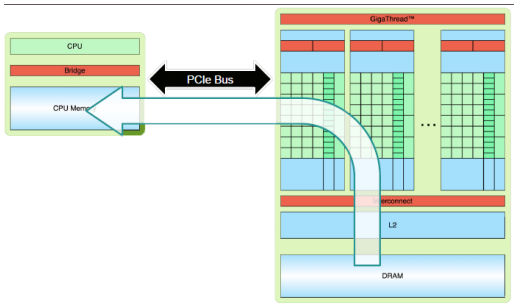


Fig. 3: Simple Processing Flow

1) Copiar dados de entrada de memória da CPU para a memória do GPU;
2) Carregar o programa para a GPU e executá-lo, fazendo uso da memória cache no *chip* para um aumento de *performance*;
3) Copiar os resultados da memória da GPU para a memória do CPU.

A Figura 4 ilustra a ordem da metodologia seguida pelo grupo no que toca às interações entre o Host (CPU) e o Device (GPU).

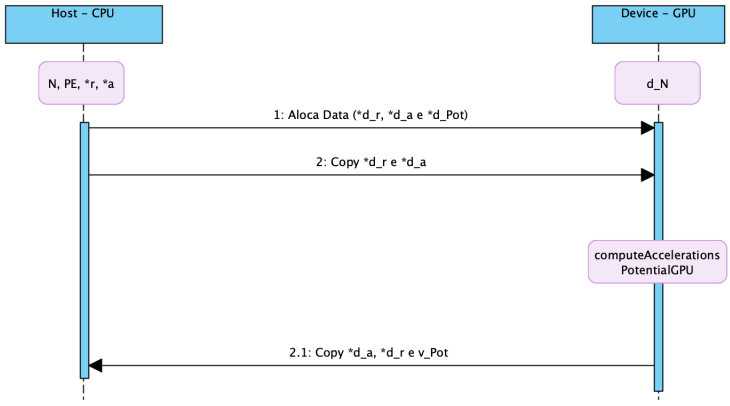


Fig. 4: Interação Host-Device

### *Primeira Tentativa*

Na primeira abordagem do grupo, após todos os dados estarem carregados em GPU foram necessárias fazer algumas alterações de maneira a otimizar e garantir a correta execução do programa.

1) Uma dessas alterações foi a criação de um *array* com memória partilhada tal como mencionado anteriormente. A vantagem obtida é que, ao carregar os valores do *array* para a memória compartilhada, os fios de execução dentro de um bloco podem aceder a esses valores mais rapidamente do que se tivessem que ler diretamente da memória global para cada cálculo subsequente. Isto é especialmente útil quando os fios de execução realizam cálculos que dependem dos mesmos dados, como é o caso deste código.
2) Após o término do ciclo, a variável local *vPot_local* é armazenada na posição *d_Pot[i]* do *array d_Pot*. Cada fio de execução calcula a sua própria contribuição para o potencial e a armazena na posição correspondente de *d_Pot*. Isto é feito de forma atómica para evitar *data races*. A soma total da energia potencial será calculada posteriormente no *host*, somando todos os elementos do *array d_Pot*.
3) Novamente, de maneira a combater as *data races* foi criada e usada a função `__device__ double atomicAddDouble(double* address, double val)` de maneira a garantir que múltiplos fios de execução não concorram simultaneamente para atualizar a mesma posição de memória (*d_a*). A operação de adição não é intrinsecamente atómica em GPUs, e em situações onde vários fios de execução podem tentar atualizar o mesmo valor ao mesmo tempo, é estritamente necessário usar uma operação atómica para garantir consistência nos resultados.

Apesar desta solução estar 100% funcional e correta, não estávamos satisfeitos com os valores obtidos, por volta de 8 segundos de tempo total e 4.5 de tempo de GPU, o que levou a pensar numa segunda alternativa, de maneira a melhorar o tempo de execução do programa.

### *Segunda Tentativa*

Na segunda abordagem, o ponto fulcral foi a remoção do uso de operações *atomicAddDouble*, que são mais lentas devido à contenção de memória partilhada. Em vez disso, utilizamos uma estratégia de redução para atualizar as variáveis globais. Por outras palavras, as acelerações calculadas (*d_a_aux*) são atribuídas diretamente aos *arrays* globais *d_a*, eliminando a necessidade de operações atómicas.

Outra melhoria foi a implementação de um segundo *array shared_Pot*, pois permite que cada fio de execução acumule localmente contribuições parciais para a energia potencial num bloco, utilizando memória partilhada. Após essa acumulação, realiza-se uma redução atómica eficiente para obter o valor total da energia potencial para o bloco. Esta abordagem minimiza operações atómicas e otimiza a comunicação entre fios de execução, resultando em melhor desempenho e eficiência na execução do código paralelo em GPUs.

Foi também necessário, a implementação de uma redução paralela hierárquica, semelhante a uma árvore invertida, de maneira a otimizar o desempenho na GPU. Essa abordagem eficiente reduz o número de operações necessárias, minimiza a sobreposição de operações entre fios de execução, e utiliza sincronização eficiente. Isto resulta em uma execução mais rápida e eficaz do algoritmo, melhorando significativamente o tempo de processamento e a utilização dos recursos da GPU. Tal como está representado pela Figura 5.

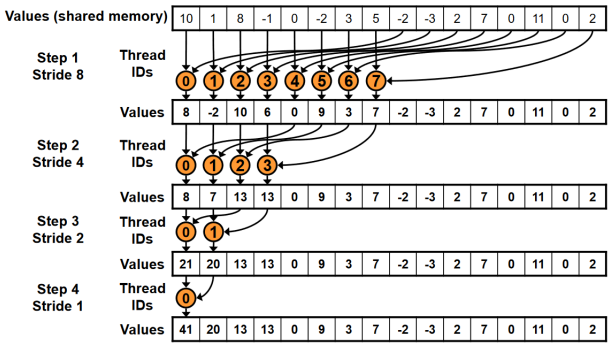


Fig. 5: Inverted Tree

### *Terceira Tentativa*

Apesar de termos verificado avanços significativos no tempo de execução da primeira tentativa para a segunda, o grupo de trabalho optou, numa fase final, por "juntar" ambas numa só. Assim, procuramos

compreender o resultado que poderiam produzir. Em outras palavras, tentamos, por um lado, manter apenas um *array __shared__*, como na primeira tentativa, e, por outro lado, evitar o uso de *atomicAddDoubles*, como na segunda tentativa. Com esta abordagem alcançamos os nosso melhores tal como é possível ver na Figura seguinte:

| Tentativa | Tempo Total (s) | Tempo GPU (s) | Tempo CPU (s) |
|---|---|---|---|
| **1** | 8.13 | 4.55 | 3.58 |
| **2** | 3.80 | 1.32 | 2.48 |
| **3** | 3.32 | 1.01 | 2.31 |

TABLE I: Tempos de Todas as Tentativas da Fase 3

## ANÁLISE COMPARATIVA ENTRE AS DIFERENTES VERSÕES DO CÓDIGO

Neste capítulo, iremos efetuar uma comparação entre as tres versões do código no que toca a várias métricas, focando a nossa atenção no tempo de execução e na escalabilidade, pois são estas as mais utilizadas para medir a performance de um programa.

*Análise do Tempo de Execução*

Na Tabela II é possível observar os valores do tempo de execução obtidos desde o código original até à Fase 3. Por outro lado, a tabela III fornece-nos os valores do tempo total, tempo de GPU e tempo de CPU, no que toca a execução do programa, obtidos na ultima fase do projeto.

| Fase | Tempo (s) |
|---|---|
| **1 - Código Original** | 236 |
| **2 - Vetorização** 1 | 4.7 |
| **3 - OpenMP** | 1.55 |
| **4 - CUDA** | 3.32 |

TABLE II: Tempos para cada Fase

| #Threads | Tempo Total (s) | Tempo GPU (s) | Tempo CPU (s) |
|---|---|---|---|
| 1 | 28.39 | 24.04 | 4.41 |
| 2 | 14.95 | 12.51 | 2.46 |
| 4 | 9.14 | 6.78 | 2.35 |
| 8 | 7.86 | 3.89 | 3.98 |
| 16 | 6.23 | 2.02 | 4.21 |
| 32 | 4.84 | 1.01 | 3.83 |
| 64 | 5.11 | 1.04 | 4.07 |
| 128 | 3.46 | 1.03 | 2.43 |
| 256 | 3.37 | 1.02 | 2.35 |
| 512 | 3.32 | 1.01 | 2.31 |
| 1024 | 5.26 | 1.17 | 4.09 |

TABLE III: Tabela com Tempo Total, Tempo GPU e Tempo de CPU

Através da análise desta tabela, conseguimos concluir que:

1) As melhorias do código ao nível de hierarquia de memória, **ILP** e vetorização foram as que contribuiriam para a diminuição mais significativa do tempo de execução, dado que a versão inicial do código efetuava bastantes operações redundantes e não tinha qualquer tipo de otimizações.
2) Em relação ao tempo de execução, a versão com **OpenMP** revela-se a melhor. Este resultado era esperado pelo grupo devido ao facto da versão com *CUDA* ter algum tempo adicional no que toca a cópia de dados e memória do *host* para o *device* e vice-versa, o que aumenta consideravelmente o tempo de execução.
3) Se analisarmos apenas o tempo de execução do **hot-spot**, que no caso do nosso programa se prende com as funções *computeAccelerationsPotential* e *computeAccelerationsPotentialGPU*, para a versão com OpenMP e CUDA, respetivamente, podemos concluir que esta função possui menor tempo de execução quando executada na versão com CUDA ($1.55 > 1.01$, conforme demonstra a tabela IV que nos apresenta a melhor versão da execução em memória partilhada, com 512 fios de execução). Estas conclusões são algo previsíveis dado que os GPU's são dispositivos especialmente projetados para lidarem com calculo paralelos intensivos.

| #Threads | Tempo Total (s) | Tempo GPU (s) | Tempo CPU (s) |
|---|---|---|---|
| 512 | 3.320 | 1.00265 | 2.31735 |

TABLE IV: Tempos Fase 3

*Análise de Acessos à Memória*

Nesta tabela podemos encontrar várias métricas que nos ajudam a entender o desempenho da GPU em termos de acesso à memória, eficiência na utilização de recursos e velocidades de leitura/escrita.

| Description | Min Value | Max Value | Avg Value |
|---|---|---|---|
| **Achieved Occupancy** | 0.244895 | 0.245036 | 0.244969 |
| **Device Memory Read Throughput** | 0.00000B/s | 3.2586MB/s | 548.51KB/s |
| **Device Memory Write Throughput** | 84.481MB/s | 87.064MB/s | 85.668MB/s |
| **L2 Throughput (Reads)** | 14.205GB/s | 14.221GB/s | 14.213GB/s |
| **L2 Throughput (Writes)** | 76.838MB/s | 76.907MB/s | 76.872MB/s |

TABLE V: Metricas de Acessos à Memória da versão em CUDA

Através da análise desta tabela podemos tirar algumas conclusões fundamentais para a compreensão do desempenho da versão do código com *CUDA*. De relembrar que estes valores foram obtidos com 512 fios de execução, que corresponde a melhor versão em termos de tempo de execução com CUDA:

1) O nível de ocupação da memória da GPU ficou bastante distante de 1, que é o valor base que indica um uso mais eficiente dos recursos. Acreditamos que esta distância se deva ao facto de cada fio de execução ter uma carga bastante baixa, tendo em conta que utilizamos um número bastante grande de fios de execução (512) quando comparado ao número de partículas (5000).
2) Em relação à velocidade de leitura/escrita do dispositivo, é possível observar que as leituras são efetuadas num tempo de execução significativamente menor do que as escritas, possivelmente devido ao facto das escritas necessitarem de operações adicionais no que toca ao controlo de concorrência.
3) Em relação à velocidade de leitura/escrita na *cache* de nivel L2, verifica-se algo curioso. A velocidade de escrita é substancialmente mais rápida do que a velocidade de leitura, o que nos leva a concluir que ou a *cache* L2 é suficientemente grande para conter a maioria dos dados ou que a percentagem de *hits* é bastante superior a percentagem de misses, o que confirma a existência de localidade espacial nos dados da memória da *cache* L2.

*Análise de Escalabilidade*

Por fim, pretendemos analisar detalhadamente a escalabilidade das duas versões paralelas desenvolvidas pelo grupo, a versão com **OpenMP** e a versão com **CUDA**.

| Threads | Real Time (s) |
|---|---|
| 1 | 29.195 |
| 2 | 14.631 |
| 4 | 7.366 |
| 8 | 3.749 |
| 16 | 1.957 |
| 20 | 1.594 |
| 32 | 1.660 |
| 40 | 1.668 |

TABLE VI: Tempos de Execução para diferentes número de *Threads*

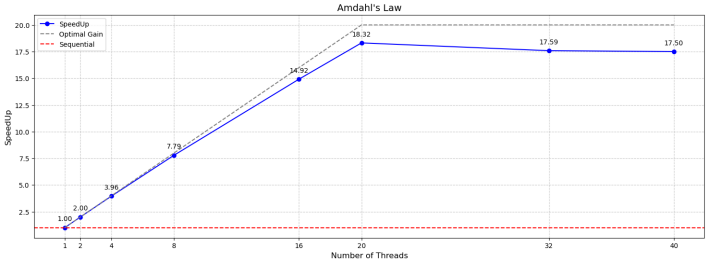


Fig. 6: SpeedUp Graph - OpenMP

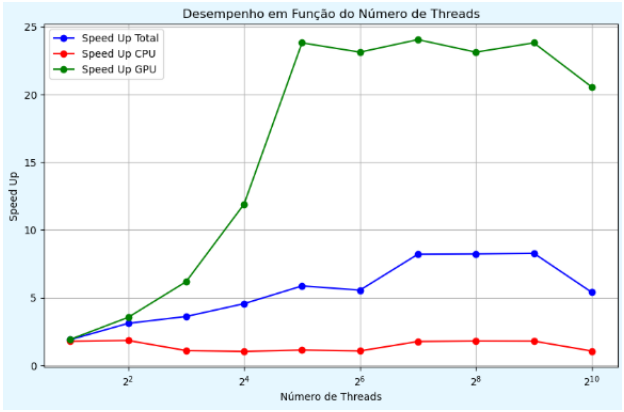


Fig. 7: SpeedUp Graph - Cuda

Através da análise da Tabela III e da Tabela VI, bem como dos seus respetivos gráficos de SpeedUp (Ganho) conseguimos concluir que:

- Tanto a versão com **OpenMp** como a versão com **CUDA** possuem uma escalabilidade forte mas não ideal, pois apresentam um aumento no ganho até um certo *threshold*, que é relativo ao dispositivo onde são executados os programas (CPU e GPU).
- Em relação a escalabilidade com a versão em **OpenMP**, podemos concluir que esta até a um threshold de 20 *threads*, que corresponde ao número máximo de núcleos do *cluster*, mantendo-se constante a partir daí, o que leva a conclusão que a competição por recursos em cada núcleo não permite que o ganho aumente.
- Em relação a escalabilidade com a versão em **CUDA**, podemos concluir que esta é conseguida principalmente com a ajuda da GPU, quando observamos o ganho do tempo total, tempo de GPU e tempo de CPU, visto que o ganho do tempo de GPU é aquele que possui um aumento mais acentuado com o aumento do número de *threads* até um *threshold* de cerca de 48 fios de execução.
- Quando comparando o ganho das duas versões, podemos observar que a *threshold* do ganho da versão de **CUDA** é superior a *threshold* do ganho com a versão de **OpenMP** (48 ¿ 20), que são ambos limitados pelo *hardware* do *cluster*. Isto leva-nos a conclusão que a versão com CUDA possui melhor escalabilidade, pois apresenta um *threshold* superior ao da versão com **OpenMP**.

## Conclusão

Em suma, ao longo de todo o semestre, aprimoramos os nossos conhecimentos relativos à vetorização na primeira fase, implementação de paralelismo através de diretivas OpenMP e, por fim, exploramos o uso de GPUs com recurso a plataforma de computação paralela *CUDA*. Esta jornada de aprendizagem proporcionou uma compreensão mais profunda e prática destes conceitos, permitindo-nos desenvolver competências sólidas e aplicáveis na otimização do desempenho computacional.