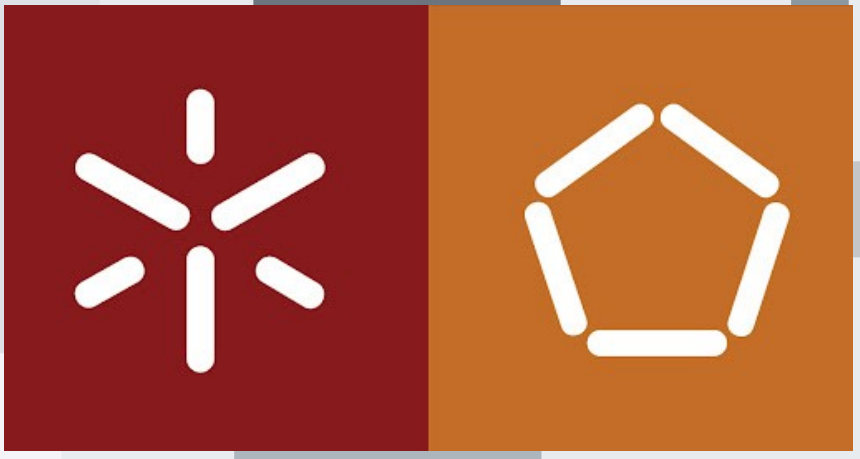


Computação Paralela

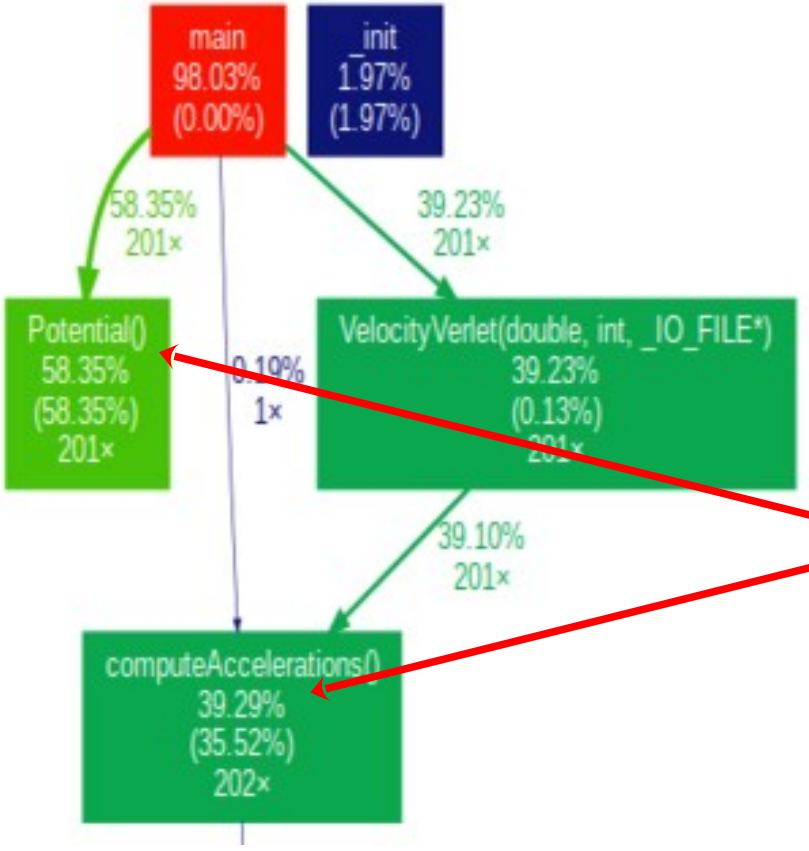
MEI - 2023/2024

Afonso Xavier Cardoso Marques - PG53601
Renato André Machado Gomes - PG54174

Trabalho Prático 1

- O principal objetivo foi avaliar os benefícios da otimização do código em uma única thread, por meio de alterações em certos algoritmos presentes no código original.
- Em resumo, foram removidos cálculos complexos de dentro de *loops* sendo substituídos por alternativas de melhor desempenho que não interferem nos valores finais de *output*.
- Foram também desconstruídos *loops* com tamanho conhecido, o que permitiu remover instruções adicionais e melhorar o desempenho, mesmo que ligeiramente.
- Realizamos testes usando a ferramenta *perf* para avaliar o desempenho da nossa versão do código.
- Por fim, os resultados obtidos foram analisados e comparados com os da versão original.

Utilizamos a ferramenta *gprof* para avaliar quais os pontos do código que consumiam mais tempo e poder computacional.



- Ao analisar o gráfico conseguimos perceber que as funções que consomem a maior parte do tempo de execução são *Potencial()* e *computeAccelerations().*
- Isto significa que são as melhores candidatas para otimização.

original.txt - Bloco de notas

Ficheiro Editar Formatar Ver Ajuda

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
58.35	17.79	17.79	201	88.51	88.51	Potential()
35.52	28.62	10.83	202	53.61	59.31	computeAccelerations()
3.77	29.77	1.15	942014880	0.00	0.00	__gnu_cxx::__promote_2<decltype (((__gnu_cxx::__promote_2<double, std::__is_integer<double>::__va
1.97	30.37	0.60				_init
0.26	30.45	0.08	1	80.00	80.00	__gnu_cxx::__promote_2<decltype (((__gnu_cxx::__promote_2<int, std::__is_integer<int>::__value>::_
0.13	30.49	0.04	201	0.20	59.51	VelocityVerlet(double, int, _IO_FILE*)
0.00	30.49	0.00	6480	0.00	0.00	gaussdist()
0.00	30.49	0.00	201	0.00	0.00	MeanSquaredVelocity()
0.00	30.49	0.00	201	0.00	0.00	Kinetic()
0.00	30.49	0.00	1	0.00	80.00	initialize()
0.00	30.49	0.00	1	0.00	0.00	initializeVelocities()
0.00	30.49	0.00	1	0.00	0.00	__gnu_cxx::__enable_if<std::__is_integer<int>::__value, double>::__type std::floor<int>(int)

% time — the percentage of the total running time of the program used by this function.

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call — the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name — the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Ln 7, Col 77 | 100% | Windows (CRLF) | UTF-8

Na função *Potential()*:

- Removeu-se a função *sqrt()*, passando a ter uma complexidade de cálculos muito mais baixa:

$$\text{rnorm} = \sqrt{r2} \qquad \text{quot} = \frac{sigma}{(\sqrt{r2})^2}$$

$$\text{quot} = \frac{sigma}{rnorm} \equiv \text{term1} = quot^6$$

$$\text{term1} = quot^{12} \qquad \text{term2} = quot^3$$

$$\text{term2} = quot^6$$

- Eliminamos a utilização da função *pow()*, substituindo-a por multiplicações.
- Remoção da condição *if (j != i)* que garantia que uma partícula não interage consigo mesma. Removendo essa condição e começando os *loops* a partir de i+1, garantimos o mesmo, mas com melhor desempenho.

Na função *computeAccelerations()*:

- Novamente, eliminamos a utilização da função *pow()*, substituindo-a por multiplicações diretas.
- Os *loops* sobre k foram descontruidos, ou seja, cada elemento é calculado separadamente, sem um loop explícito, o que leva a um desempenho ligeiramente melhor.

Resultados do TP1:

TABLE I
TABLE WITH EXECUTION TIME FOR 3 ITERATIONS OF ORIGINAL VERSION OF MD

Iteration	Texe
1	236,95 sec
2	236.79 sec
3	236,92 sec

TABLE II
TABLE WITH EXECUTION TIME FOR 3 ITERATIONS OF OPTIMIZED VERSION OF MD

Iteration	Texe
1	6.82 sec
2	6.43 sec
3	6.65 sec

TABLE III
TABLE WITH METRICS FOR EACH VERSION OF MD

Version MD.cpp	#I	Average #CC	Average CPI	Average Texe
Original	1.243.580.424.482	780.836.157.356	0,6	236,8770 sec
Optimized	35.346.319.145	21.426.058.674	0,6	6,654 sec

Trabalho Prático 2

- O principal objetivo foi avaliar os benefícios da otimização do código com múltiplas *threads*, utilizando a ferramenta OpenMP para paralelizar a execução do código MD.
- O número estático de partículas, N, aumentou de 2160 para 5000.
- Foi feita uma análise de quais instruções potenciais do OpenMP poderíam ser usadas para melhorar a paralelização com base em experiências anteriores realizadas nas aulas práticas do curso.
- Testamos diferentes diretivas *pragma* nos vários *loops* contidos nas funções *Potential* e *computeAccelerations* analisando os resultados de desempenho com o *perf*.
- Alternamos o número de *threads* em execução para determinar qual o melhor valor de *speedup*.
- Os resultados obtidos foram analisados e comparados com os resultados da versão sequencial do TP1.

Na função *Potential()*:

```
#pragma omp parallel for reduction(+:Pot)
```

- Indica ao compilador para paralelizar o ciclo de forma a que cada iteração seja executada por uma *thread* diferente, simultaneamente criando uma cópia local da variável 'Pot' para cada *thread*, combinando-as no final da execução.

```
#pragma omp simd reduction(+:r2)
```

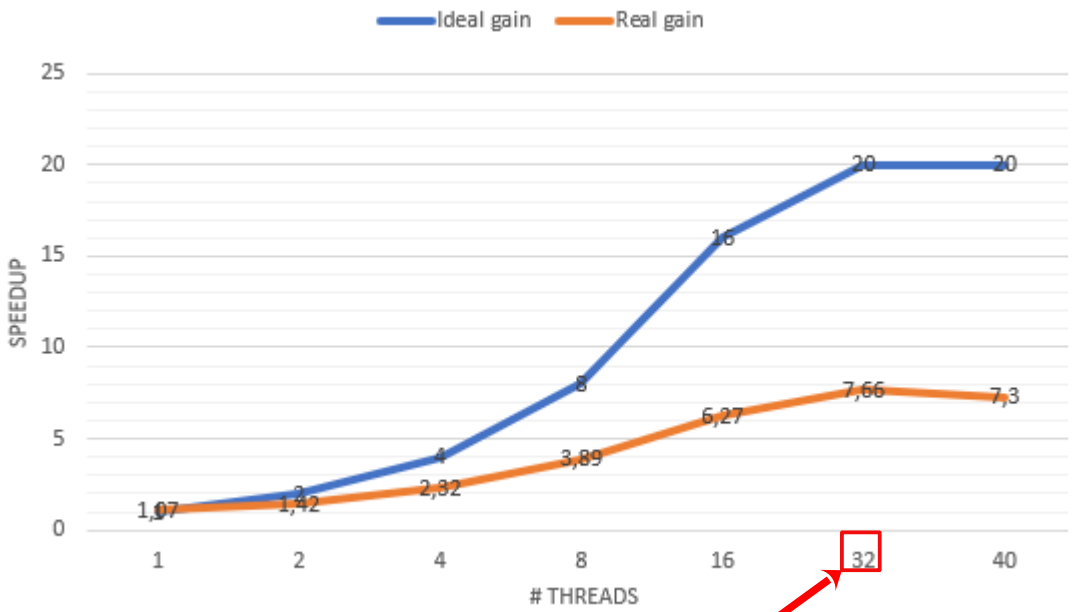
- Indica ao compilador para vetorizar o ciclo fazendo uma cópia local da variável 'r2' para cada *thread,* combinando tudo no final.

Na função *computeAccelerations()*:

```
#pragma omp parallel for private(j, f, rSqd, rij)
```

- Resumindo, esta diretiva paralelizará o ciclo, com cada thread a ter a sua própria cópia privada das variáveis 'j', 'f', 'rSqd' e 'rij', o que evitará, teóricamente, *data races*.

Resultados do TP2 original:



$$speedup = \frac{bestSeqTime}{parExecTime}$$

- Apesar de conseguirmos um *speedup* bastante bom com 32 *threads*, houve ocorrências de *data races* que não detetamos no momento da entrega do trabalho.
- Estas foram corrigidas na entrega final mas com valores de performance piores.

Função computeAccelerations_Potencial*() (versão corrigida WA2)*:

- Esta função é uma junção das funções *computeAccelerations* e *Potencial o que permitiu aproveitar o facto de ambas percorrerem os mesmos espaços de memória.*

- *Foi criada uma nova instrução de OpenMP:*

```
#pragma omp parallel for private(j, f, rSqd, rij) reduction(+:Pot, a[:N*3]) schedule(dynamic,48)
```

- Que se divide nas seguintes diretivas:

  - parallel for: indica que o *loop* seguinte deve ser executado em paralelo;

  - private(j, f, rSqd, rij): especifica que cada *thread* deve ter sua própria cópia dessas variáveis;

  - reduction(+:Pot, a[:N*3]): é usada para realizar operações de redução, como a soma. No nosso caso, realizamos uma redução nas variáveis 'Pot' e 'a', onde 'Pot' é um escalar e 'a' é uma matriz tridimensional. Isso significa que cada *thread* mantém sua própria cópia de 'Pot' e 'a', e no final do *loop*, essas cópias são combinadas de acordo com a operação de soma.

  - schedule(dynamic,48): controla como as iterações do *loop* são distribuídas (de forma dinâmica) entre as *threads*. Cada *thread* receberá um bloco de 48 iterações por vez até que todas as iterações sejam concluídas.

Trabalho Prático 3

- O principal objetivo foi aprimorar ainda mais o desempenho do código usando uma das três alternativas: melhorar a paralelização existente com OpenMP, usar o módulo MPI ou usar o módulo CUDA. Optamos por usar o CUDA.
- Mantivemos a junção das funções *Potential* e *computeAccelerations.*
- Foi o mais eficaz e gratificante em termos de resultados, alcançando uma média de 4 a 5 segundos de tempo de execução sem sinais de *data races* e com N definido como 5000.
- Melhorou significativamente o desempenho da simulação, utilizando o poder de processamento paralelo das GPUs. Essa melhoria permite uma computação mais eficiente das interações e dinâmicas das partículas, embora também introduza complexidade adicional em termos de gestão de memória e arquitetura do programa.

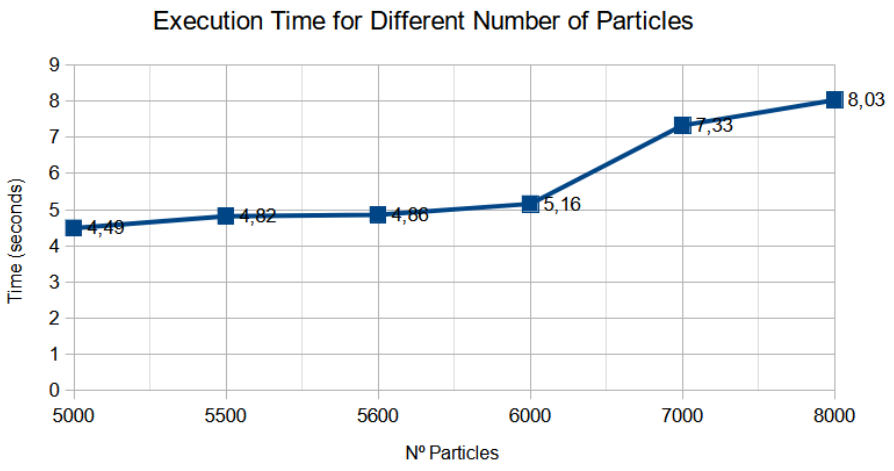
Função computeAccelerations_Potencial*()*:

- Incluí as operações para alocação de memória na GPU e para transferência de dados entre a CPU e a GPU. Por exemplo, a memória é alocada para as posições das partículas (r), acelerações (a) e energia potencial (PE), e os dados são transferidos entre o hospedeiro (CPU) e o dispositivo (GPU). Este aspecto é crucial porque a GPU necessita do seu próprio espaço de memória e não pode aceder diretamente à memória da CPU.

Função computeAccelerations_Potencial_KERNEL*()*:

- Resumindo, contem as funções do Kernel. É executada na GPU e é responsável por calcular as acelerações e energia potencial. A função utiliza as capacidades de processamento paralelo do CUDA, com cada *thread* (indexada por i) a lidar com cálculos para uma partícula diferente na simulação. É afetada pelo número de *threads* por bloco que é alocada ao programa.

Resultados do TP3:



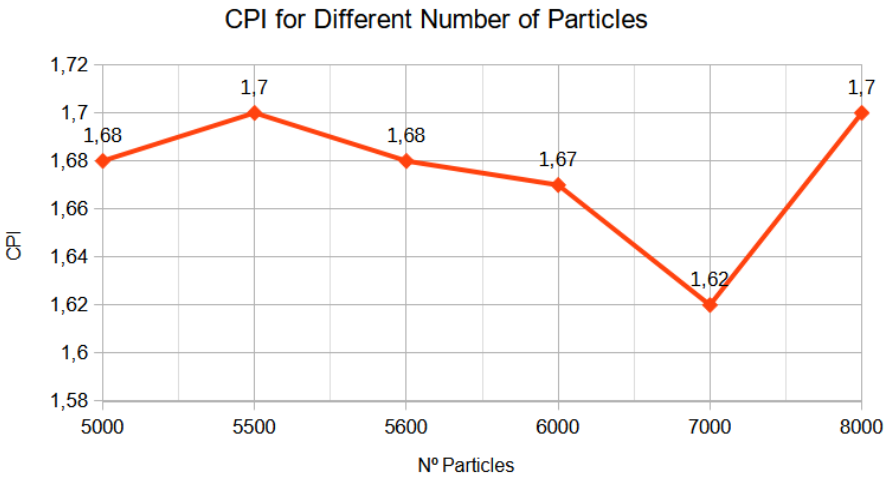
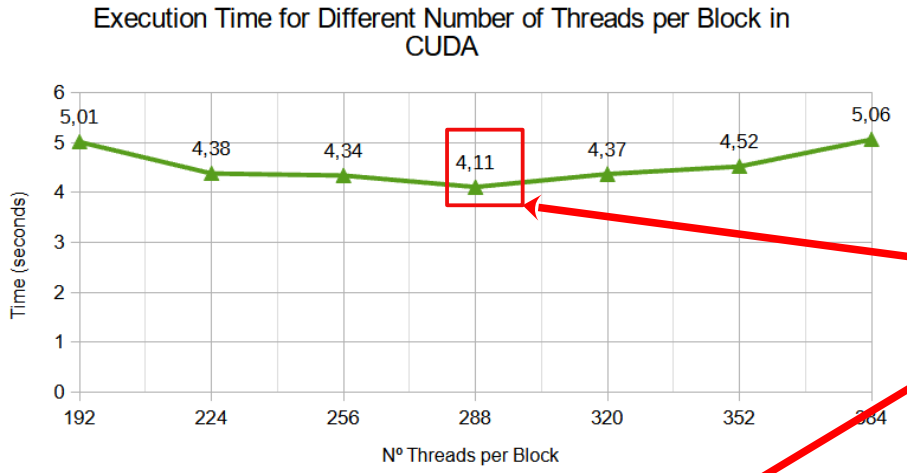


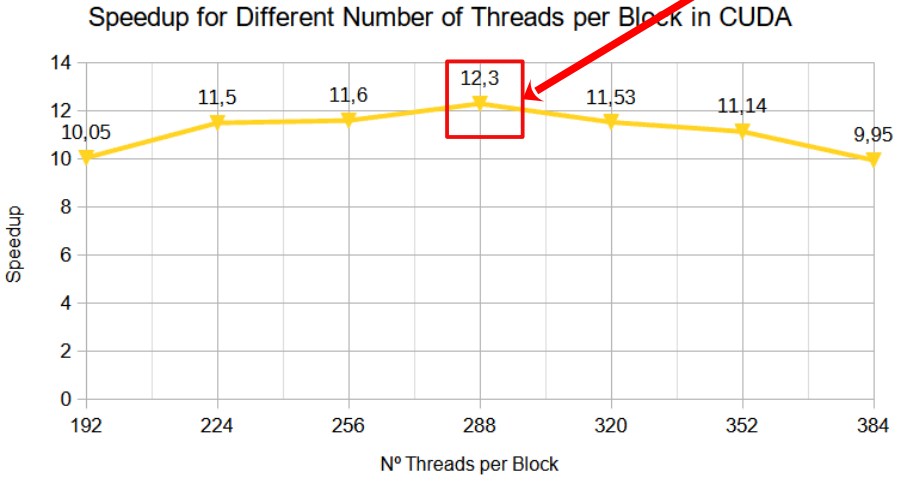
TABLE I

TABLE WITH EXECUTION TIME OF *MDpar_CUDA* IN MULTIPLE RUNS

#Run	Execution Time
1	0 m 4.11 s
2	0 m 4.16 s
3	0 m 4.18 s
4	0 m 4.06 s
5	0 m 4.17 s

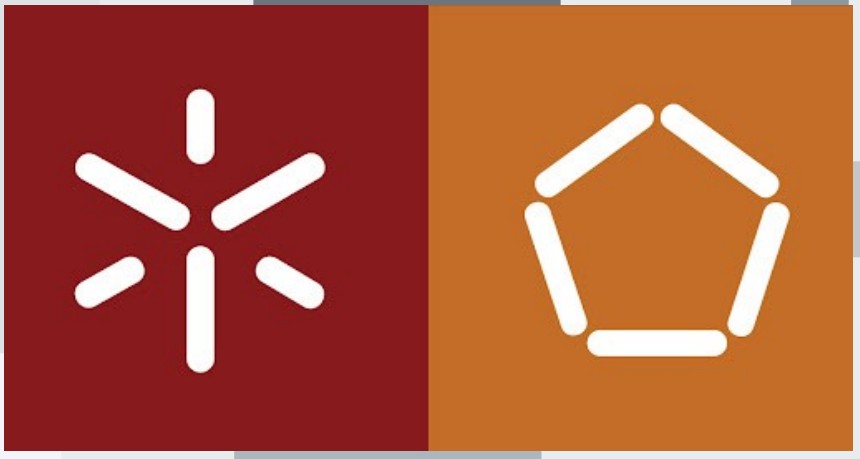
Resultados do TP3:





- Ao analisar os gráficos conseguimos perceber que o melhor tempo e *speedup* são obtidos se usarmos 288 *threads* por bloco dentro do *kernel* em CUDA.

$$speedup = \frac{bestSeqTime}{CUDAExecTime}$$



Computação Paralela

MEI - 2023/2024

Afonso Xavier Cardoso Marques - PG53601
Renato André Machado Gomes - PG54174