Computação Paralela

Grupo pg54123_pg53645

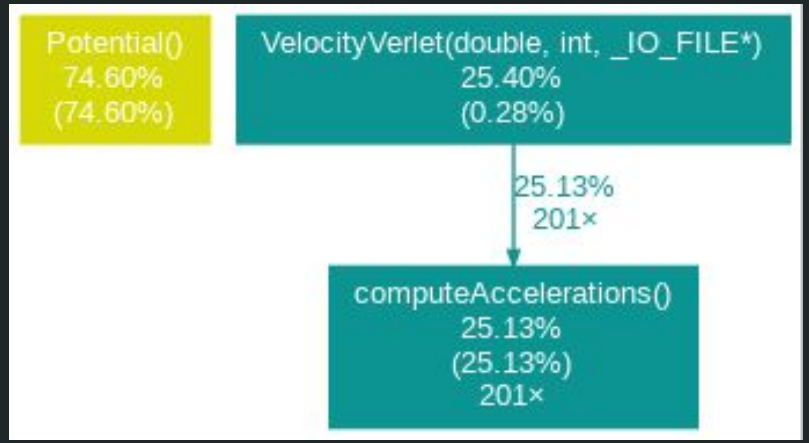
Ana Rita Santos Poças, PG53645
Orlando José da Cunha Palmeira, PG54123

WA1

Optimizações ao código sequencial

Análise e perfil do código

- Potential: aproximadamente $O(N^2)$, complexidade quadrática
- computeAccelerations: aproximadamente $O(N^2)$, complexidade quadrática
- Existência de duas funções (`Potential` e `computeAccelerations`) que ocupam perto de 100% do tempo de execução total do programa.



Resultados obtidos com o Gprof

Optimizações aplicadas

- Alterações ao algoritmo
  - Simplificação matemática dos cálculos de ambas as funções (reduz número de multiplicações, remoção de sqrt, pow, etc...)
  - Redução do número de iterações do *loop* da Potential para metade
  - Fusão das funções `Potential` e `computeAccelerations` para evitar cálculos repetidos
- ILP
  - Remoção de *loops* com número fixo de iterações (3)
- Hierarquia de memória
  - Uso de variáveis temporárias no ciclo interno da função computeAccelerations para acumular valores da matriz "a", aproveitando a constância do índice "i" para optimizar o acesso à memória, tirando partido da **localidade temporal**.
- Vectorização
  - Utilização das flags “`-ftree-vectorize`” e “`-msse4`”
  - Remoção do “if” do *loop* nos cálculos do potencial

Resultados

N = 2160	
Versão não optimizada	Versão optimizada
Tempo de execução: ~200 segundos Número de instruções: ~1.3×10^{12} CPI: ~0.6 L1 Cache misses (load + store): ~3×10^{9}	Tempo de execução: ~3.4 segundos Número de instruções: ~1.93×10^{10} CPI: ~0.6 L1 Cache misses (load + store): 4.8×10^{8}

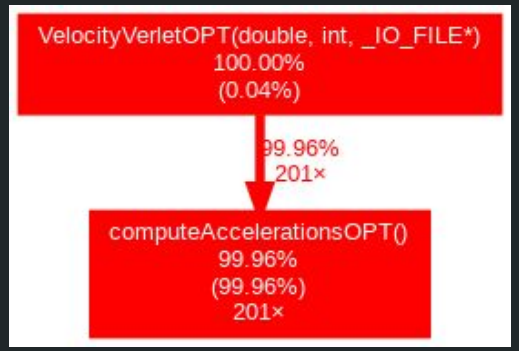
Speedup: ~58.82
Redução do número de instruções: ~67,36
Redução do CPI: 1 (sem mudança no CPI)
Redução de Cache Misses: ~6,25

WA2

Exploração de paralelismo com memória partilhada (OpenMP)

Análise e perfil do código

- Função computeAccelerations ocupa 99.96% do tempo de execução total do programa.



Resultados obtidos com o Gprof

- Dois **ciclos for** aninhados (complexidade quadrática de aproximadamente $O(N^2)$) onde todos os cálculos são realizados. As restantes operações apenas **inicializam valores** na memória.
- As técnicas de paralelismo serão então aplicadas no *outer loop* (variável i) da função computeAccelerations onde são realizadas as operações matemáticas.

Construção da directiva OpenMP

- Paralelizar o *outer loop*

```
#pragma omp parallel for
```

- Protecção de variáveis sensíveis a escrita concorrente

```
#pragma ...  private(ai0,ai1,ai2,rij,rSqd,r2,rSqd3,rSqd7,f,term1,term2,rijf)
```

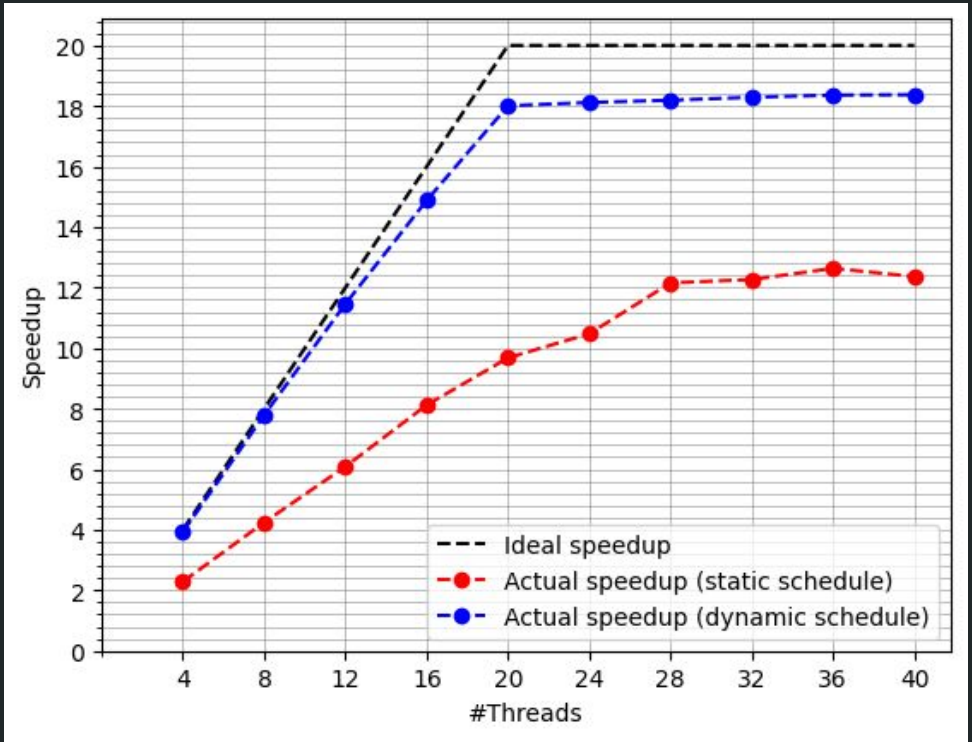
- Operações de redução na variável **Pot** e na matriz **a** (evita atomic's e critical's)

```
#pragma ... reduction(+:Pot,a[:N][:3])
```

- Distribuição de carga equitativa entre *threads* (agendamento dinâmico)

```
#pragma ... schedule(dynamic)
```

Resultados



Speedup relativamente à versão sequencial optimizada (escalabilidade forte)

N = 5000, 20 cores	
Versão sequencial	Versão paralela (40 *threads*)
~23 segundos	~1.3 segundos

Comparação do tempo de execução entre as implementações

Resultados

O *speedup* ideal não foi atingido por algumas razões:

- **Redução** da matriz **a** (dimensões N × 3)
- Algoritmo limitado pela **largura de banda de memória**: Existem vários acessos à memória, nomeadamente à matriz **a**.
- ***Overhead* de gestão de *threads*** devido à utilização de '`schedule(dynamic)`'

WA3

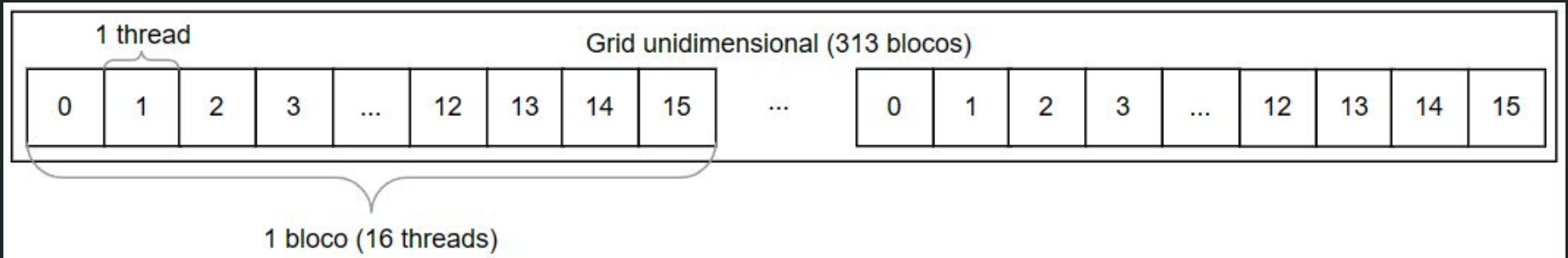
Exploração de paralelismo com aceleradores (CUDA)

(feita com base na análise e perfil do código feita na segunda fase)

Implementação em CUDA

- *Launcher* do *kernel*
  - Aloca a memória na GPU para os arrays relativos às matrizes `a`, `r` e ao *array* para o cálculo do potencial.
  - O *array* para o cálculo do potencial permite o cálculo **paralelo** do potencial para cada *thread* sem a necessidade de controlo de concorrência já que cada *thread* escreve na sua própria posição desse *array*.
  - Realiza a cópia dos dados das matrizes `a` e `r` do *host* para a GPU antes da execução do *kernel* e recupera os dados da matriz `a` e do *array* do potencial da GPU de volta para o *host*.
- *Kernel*
  - Cálculos e lógica executados de modo muito semelhante à versão sequencial.
  - *Outer loop* inexistente, uma vez que o *kernel* representa uma iteração desse *loop* executada por cada *thread*.
  - Operações atómicas nas escritas na matriz `a` de forma a evitar *data races*.

Implementação em CUDA

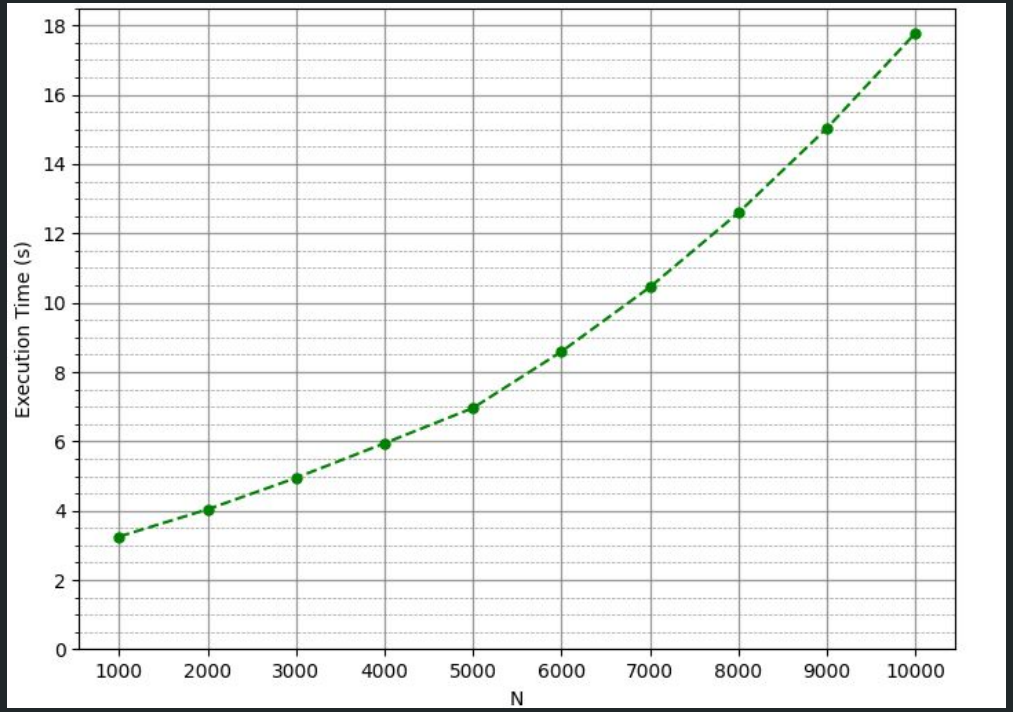


Grid CUDA utilizada

Resultados

N = 5000	
Threads por bloco	Tempo de execução total (s)
16	6.432
32	7.819
64	7.812
128	7.834
256	8.228
512	9.213
1024	14.019

Tempo de execução em função da dimensão dos blocos



Análise de escalabilidade fraca (16 threads/bloco)

Análise crítica da implementação com CUDA

- A carga de uma *thread* depende directamente do seu id (maior o id ⇒ menos carga da *thread*) devido ao loop no kernel. Existe um **desbalanceamento de carga**.
- Sucessivas **transferências de dados** de e para a GPU em cada execução do kernel constituem um **gargalo de desempenho**
- **Operações atómicas** no *kernel* constituem um **gargalo de desempenho**.

Operation	Total time	Calls	Average
kernel	4.55142s	202	22.532ms
memcpy HtoD	8.8808ms	404	21.982us
memcpy DtoH	5.0837ms	404	12.583us
cudaMemcpy	4.60259s	808	5.6963ms
cudaMalloc	282.32ms	606	465.88us

N = 5000, 16 threads/bloco

Comparação das versões

- A versão que apresentou **melhores resultados** foi a implementação com **OpenMP** (N = 5000 ⇒ ~1.3s).
- Em seguida, temos a versão com **GPU's** (N = 5000 ⇒ ~6.432s).
- E por último, a **versão sequencial** apresenta a **pior performance** (N = 5000 ⇒ ~23s)**.**

Computação Paralela

Grupo pg54123_pg53645

Ana Rita Santos Poças, PG53645
Orlando José da Cunha Palmeira, PG54123