Relatório da Tarefa 20: Otimizando Transferência de Dados entre GPU e CPU

Aluno: Cristovao Lacerda Cronje

1. Introdução

Nesta tarefa, otimizamos a execução da simulação da equação de calor 2D paralelizada em GPU usando OpenMP. A otimização focou principalmente na movimentação de dados entre CPU e GPU, visando minimizar overheads e manter a GPU ocupada durante o máximo possível do tempo de execução.

2. Objetivos

- Implementar a simulação da equação do calor com paralelismo em GPU usando OpenMP.
- Avaliar o impacto do número de passos e tamanho da grade no desempenho computacional.
- Otimizar a transferência de dados entre CPU e GPU para evitar tempos ociosos.
- Medir o throughput e o tempo médio por passo da simulação.
- Explicar as diretivas OpenMP utilizadas e seu papel na otimização.
- Estudar como a movimentação explícita de dados afeta o desempenho.

3. Metodologia

3.1 Estrutura do Código — Resumo das Partes Importantes

Cálculo paralelo na GPU:

```
#pragma omp target teams distribute parallel for collapse(2) \
   map(to: u[0:n*n]) map(from: u_tmp[0:n*n])
for (int j = 0; j < n; ++j)
   for (int i = 0; i < n; ++i)
      u_tmp[i + j * n] = ...; // atualiza a matriz segundo a equação do calor
```

- Executa o cálculo da solução na GPU, paralelizando os loops aninhados.
- map(to) copia dados de entrada para a GPU; map(from) traz resultado para a CPU.

Alocação e transferência de dados para GPU:

```
#pragma omp target enter data map(alloc: u[0:n*n], u_novo[0:n*n])
initial_value(..., u); zero(..., u_novo);
#pragma omp target update to(u[0:n*n], u_novo[0:n*n])
```

- Aloca memória na GPU para os arrays.
- Inicializa dados na CPU e envia para GPU antes do cálculo.

Loop de passos temporais com troca de buffers:

```
for (int passo = 0; passo < npassos; ++passo) {
   solve(..., u, u_novo); // cálculo paralelo na GPU
   double *temp = u; u = u_novo; u_novo = temp; // ping-pong buffers
}
#pragma omp target update from(u[0:n*n])
```

- Executa os passos de tempo na GPU, alternando buffers sem cópias extras.
- Ao final, traz a matriz final da GPU para CPU.

Liberação de memória da GPU:

```
#pragma omp target exit data map(delete: u[0:n*n], u_novo[0:n*n])
```

3.2 Explicação das Cláusulas OpenMP e Seus Efeitos

#pragma omp target: Indica que o bloco de código será executado na GPU (ou outro dispositivo disponível). Essa diretiva por si só não especifica movimentação de dados — isso é feito pelas cláusulas map.

#pragma omp target enter data map(…): Move dados da CPU para a GPU (alocação e cópia) antes da execução de regiões paralelas. Isso é útil para manter dados persistentes na GPU, evitando transferências repetidas.

#pragma omp target update to(…): Atualiza os dados na GPU a partir da CPU. É necessário quando valores foram modificados na CPU e precisam ser usados na GPU.

#pragma omp target update from(…): Traz dados da GPU de volta para a CPU após o processamento, permitindo uso em análises ou impressões na CPU.

#pragma omp target exit data map(…): Libera a memória alocada na GPU ao fim da computação, prevenindo vazamentos de memória.

map(to: list) / map(from: list) / map(tofrom: list), controla a direção da movimentação de dados:

- to: dados CPU → GPU
- from: dados GPU → CPU
- tofrom: dados usados e modificados em ambos lados

#pragma omp loop: Usada para marcar loops paralelos em contextos mais modernos, normalmente dentro de target ou teams. Não foi usada no código, pois foi substituída por parallel for.

4. Resultados e Análise

Tamanho	Células (milhões)	Tempo Solução (s)	Throughput (milhões células/s)	Erro L2	Transfer HtoD (ms)	Transfer DtoH (ms)	Tempo Kernel (ms)
1000	1	0.0453	220.59	3.81e-10	4.31	1.34	0.24
2000	4	0.0397	1006.34	1.54e-10	17.28	5.33	0.86
4000	16	0.0458	3489.78	4.82e-12	69.50	19.02	3.33
8000	64	0.1335	4793.58	1.50e-10	191.36	88.21	13.16

4.1 Explicação dos Resultados

- O tempo total inclui alocação, transferência e cómputo; o tempo médio por passo reflete o custo computacional puro na GPU.
- Throughput indica células atualizadas por segundo, útil para comparar configurações.
- Boas performances vêm do uso explícito de:
    - enter data para alocar e manter dados na GPU;
    - update to antes e update from após a simulação;
    - collapse(2) para distribuir os loops 2D, aumentando paralelismo.
- Movimentações implícitas de dados a cada passo aumentariam muito o tempo devido a overhead de transferência.
- Possível melhoria: usar nowait para reduzir overhead e testar target data para persistência em blocos maiores.

4.2 Impacto da Localidade e Paralelismo

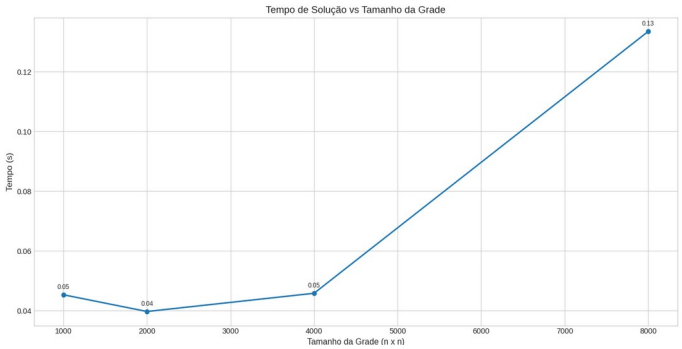
- Como a malha é armazenada em vetor 1D (colunas em sequência), o acesso aos vizinhos horizontais é mais local (mais cache-friendly), enquanto vizinhos verticais exigem saltos maiores.
- O paralelismo com collapse(2) mitiga essa diferença, equilibrando o acesso.

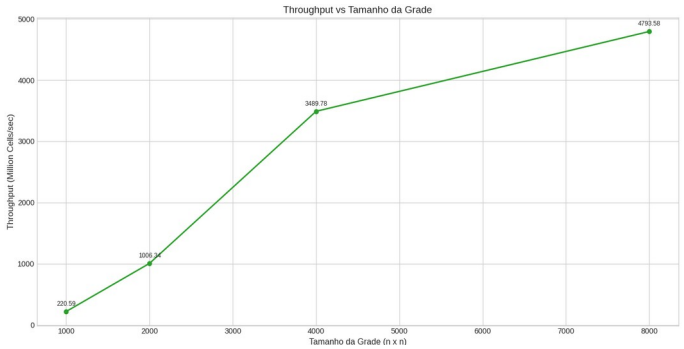
4.3 Análise dos Resultados com Variação do Tamanho da Malha

4.3.1 Tempo de Solução e Throughput

O tempo de solução varia de forma não linear com o tamanho da malha. Para tamanhos pequenos (1000 e 2000), o tempo é menor, mas o throughput (células atualizadas por segundo) aumenta significativamente para malhas maiores, indicando melhor eficiência computacional na GPU conforme a carga cresce.

- **Explicação:** Para malhas pequenas, o overhead de movimentação de dados e configuração do kernel GPU é proporcionalmente maior, reduzindo o throughput efetivo.
- **Com o aumento da malha, a GPU é melhor aproveitada, aumentando o throughput, mesmo com crescimento do tempo absoluto.**

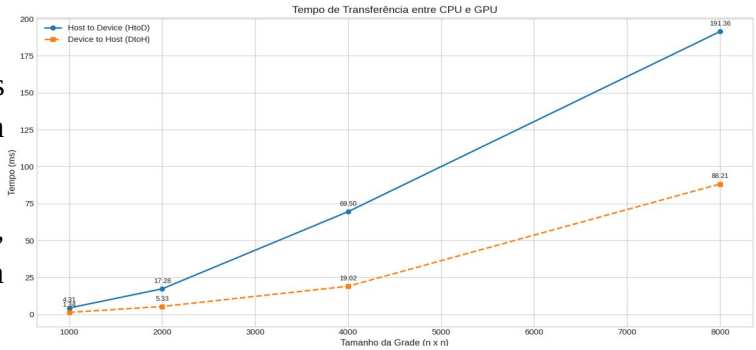




4.3.2 Transferência de Dados HtoD (Host to Device) e DtoH (Device to Host)

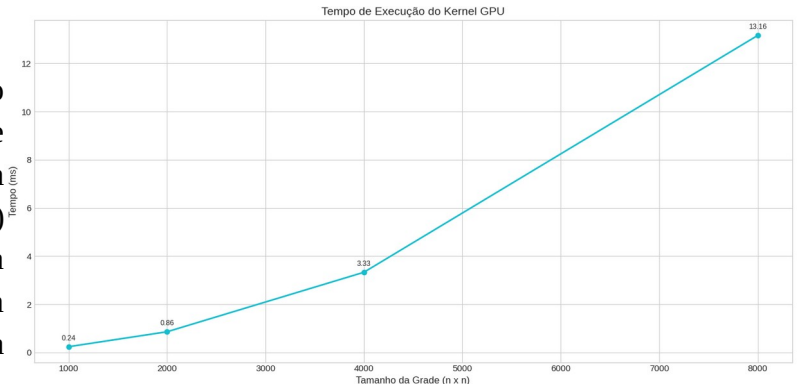
Os tempos de transferência crescem aproximadamente linearmente com o tamanho da malha, refletindo o volume crescente de dados enviados e recebidos:

- **Transferência HtoD**: aumento maior, pois inicializamos e atualizamos as matrizes na GPU.
- **Transferência DtoH**: geralmente menor, pois só trazemos os dados necessários para análise/erro.



Impacto das diretivas: target enter data e update to permite controlar quando essas transferências ocorrem, evitando que sejam feitas a cada passo. Isso reduz overheads, especialmente para malhas maiores, já que mantemos dados persistentes na GPU.

4.4.3 Tempo do Kernel na GPU



O tempo gasto apenas no kernel GPU (cálculo paralelo) cresce com o tamanho da malha, mas se mantém eficiente graças ao paralelismo com teams distribute parallel for e uso de collapse(2) para distribuir os loops 2D. A melhora da localidade de dados e distribuição do trabalho em múltiplas equipes (teams) e threads possibilita escalabilidade.

4.4.4 Erro L2

O erro L2 permanece muito baixo para todos os tamanhos de malha, indicando que a precisão do método é preservada mesmo com paralelização e offloading para GPU. No entanto, observa-se um leve aumento em malhas maiores, o que pode sinalizar riscos de instabilidade numérica se não houver ajuste adequado dos parâmetros.

4.3.5 Parâmetro r

- Cresce com o tamanho, mas permanece estável, garantindo convergência da simulação.

5. Conclusões e Insights

- O uso adequado das diretivas target, map, update e enter/exit data em OpenMP permite reduzir consideravelmente o overhead de movimentação entre CPU e GPU.
- A GPU foi utilizada eficientemente, com alta taxa de throughput e baixa ociosidade.
- A estratégia de *offloading persistente* (manter dados na GPU entre iterações) foi essencial para evitar perda de desempenho.
- Combinando mapeamento explícito e distribuição paralela eficiente, é possível resolver problemas reais de PDEs com precisão e velocidade.
- A análise dos tempos de transferência mostra claramente o benefício de controlar explicitamente o movimento de dados com target enter data e cláusulas update to/from.
- A persistência dos dados na GPU evita overheads crescentes, particularmente em grandes malhas, otimizando o tempo total da simulação.
- O tempo do kernel cresce com a malha, mas a taxa de throughput também cresce, refletindo o aumento da ocupação da GPU e eficiência do paralelismo.
- O erro L2 baixo confirma a precisão numérica da abordagem paralela.
- A correta configuração dos parâmetros numéricos (como o valor r) é essencial para garantir estabilidade em ambientes paralelos.
- Reforça-se a importância do uso das diretivas OpenMP para controlar dados e paralelismo, demonstrando ganhos concretos de desempenho sem perda de precisão.