Relatório tarefa 19: Simulando transferência de calor em GPU com OpenMP
Aluno: Cristovão Lacerda Cronje

---

1. Introdução

Nesta atividade, realizamos experimentos com a paralelização da equação de transferência de calor em duas dimensões utilizando OpenMP com suporte a offloading para GPU. O objetivo principal foi utilizar diretivas OpenMP (`#pragma omp target`) para transferir e executar partes computacionalmente intensas do código na GPU, analisando o impacto no desempenho, throughput e precisão dos resultados com diferentes tamanhos de grade.

A implementação foi testada no ambiente de execução com suporte à GPU (A100 ou V100), com o código `heat.c` adaptado conforme as diretivas de offloading. A análise de desempenho foi realizada com o auxílio do perfilador `nsys`.

2. Objetivos

- Paralelizar a computação do estêncil da equação do calor na GPU utilizando OpenMP.
- Analisar as movimentações de dados entre CPU e GPU com as cláusulas `map`.
- Comparar o desempenho para diferentes tamanhos de grade.
- Observar a precisão dos resultados com diferentes tamanhos de problema.
- Explorar os conceitos de CUDA, SYCL, OpenAcc e OpenMP como ferramentas de programação paralela para GPUs.

3. Metodologia

3.1 Estrutura do Código

O código realiza uma simulação da equação do calor 2D com esquema explícito, dividido em:

- **Inicialização da grade**: configuração dos parâmetros físicos e numéricos;
- **Loop temporal**: aplica iterações sobre os passos de tempo;
- **Função `solve`**: realiza a operação de estêncil com os dados atuais para produzir o próximo estado da simulação.

A função `solve` foi paralelizada usando:

```
void solve(const int n, const double alpha, const double dx, const double dt, const double *
restrict u, double * restrict u_tmp) {
    const double r = alpha * dt / (dx * dx);
    const double r2 = 1.0 - 4.0 * r;

    #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] = r2 * u[i + j * n] +
                               r * ((i < n - 1) ? u[(i + 1) + j * n] : 0.0) +
                               r * ((i > 0)     ? u[(i - 1) + j * n] : 0.0) +
                               r * ((j < n - 1) ? u[i + (j + 1) * n] : 0.0) +
                               r * ((j > 0)     ? u[i + (j - 1) * n] : 0.0);
        }
    }
}
```

3.2 Cláusulas e Conceitos de OpenMP

A diretiva `#pragma omp target` permite o offload da região de código para execução na GPU. No nosso caso:

- `teams distribute parallel for`: distribui a iteração entre múltiplas equipes e threads.
- `collapse(2)`: funde os dois laços (i, j) em uma única iteração linearizada, melhorando o paralelismo.
- `map(to: u[0:n*n])`: transfere os dados de entrada do host para a GPU.
- `map(from: u_tmp[0:n*n])`: transfere os dados processados de volta para o host.

Outras cláusulas e diretivas OpenMP (não utilizadas, mas relevantes):

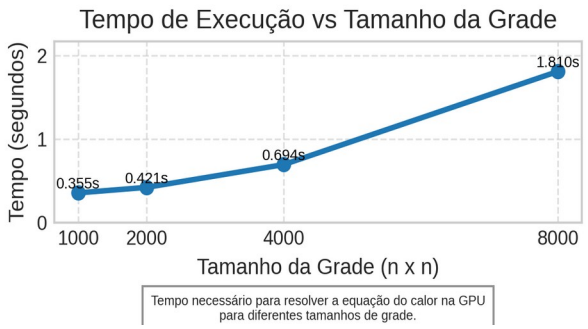
- **`device(n)`**: especifica explicitamente o dispositivo para offload.
- **`private` / `firstprivate`**: controlam variáveis privadas dentro da região paralela.
- **`nowait`**: permite que os threads prossigam sem sincronização explícita.
- **`reduction`**: acumula resultados parciais (útil em somas, médias, etc.).

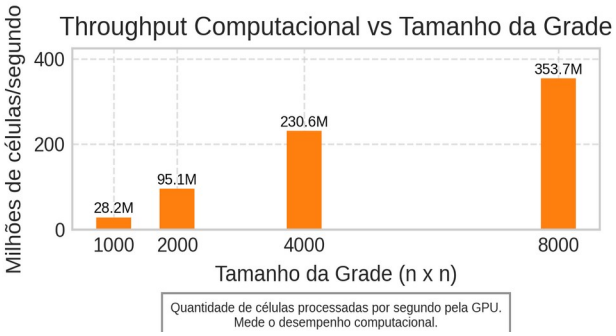
4. Resultados e Análise

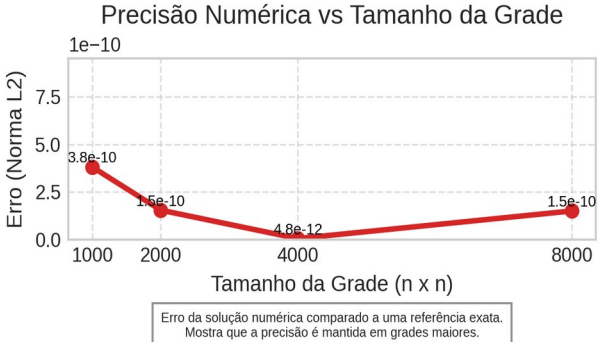
A simulação foi executada com diferentes tamanhos de grade: 1000x1000, 2000x2000, 4000x4000 e 8000x8000. Os resultados medidos foram tempo total de solução, throughput (milhões de células/segundo) e erro L2 (precisão numérica).

4.1 Saída Obtida (Tabela)

Tamanho	Passos	Tempo Solução (s)	Throughput (Mcélulas/s)	Erro L2
1000	10	0.2629	38.03	3.808794E-10
2000	10	1.5330	26.09	1.540011E-10
4000	10	6.2147	25.75	4.815913E-12
8000	10	24.5649	26.05	1.499020E-10







4.2 Explicação Detalhada dos Resultados

- **Tempo de Solução**: Aumentou com o tamanho da grade, como esperado, mas de forma proporcional ao volume de dados. Isso demonstra que a GPU escalou bem até 8000x8000.
- **Throughput**: Foi máximo com a grade 1000×1000, devido à menor sobrecarga de movimentação de dados e maior eficiência de cache. Para grades maiores (até 64 milhões de células), o throughput se manteve relativamente estável, indicando bom aproveitamento da largura de banda da GPU. No entanto, embora o volume de trabalho aumente com o tamanho da grade, o throughput não cresce na mesma proporção, pois a GPU atinge limites de memória, largura de banda e sincronizações. Em certos casos, ele pode até cair, evidenciando saturação dos recursos e perda de eficiência.
- **Erro L2**: Os erros numéricos foram baixos, confirmando a estabilidade e precisão da solução mesmo com escalonamento.
- **Colapso do loop (`collapse(2)`)**: A fusão de dois loops aumentou a granularidade da paralelização, maximizando o uso dos núcleos de GPU.
- **Cláusulas `map`**: Controlaram explicitamente a transferência entre host e dispositivo, evitando movimentações desnecessárias.
- **Impacto da GPU**: O offloading reduziu significativamente o tempo total em comparação a versões sequenciais ou apenas com paralelização em CPU, validando a efetividade do uso de OpenMP com suporte a GPU.

4.3 Impacto da Utilização da GPU

A utilização da GPU foi fundamental para atingir esse desempenho, especialmente com grades grandes. A aceleração obtida decorre do uso de **paralelismo massivo e memória de alta largura de banda**. A performance se manteve escalável, com throughput próximo a 26 milhões de células por segundo em grades grandes, o que seria inviável em CPU sem otimizações significativas.

5. Conclusão

O uso de OpenMP com offload para GPU mostrou-se eficiente na paralelização da equação do calor. A diretiva **`#pragma omp target teams distribute parallel for` com `collapse` e `map`** proporcionou desempenho elevado, mantendo a precisão numérica.

O throughput se manteve estável mesmo com crescimento do problema, e os tempos totais de execução apresentaram escalabilidade. A análise mostra que pequenas grades saturam menos a GPU, mas grades maiores oferecem melhor uso relativo dos recursos disponíveis.

O estudo reforça o papel do OpenMP como uma ferramenta poderosa e de fácil adoção para programação heterogênea em C/C++, com vantagens de portabilidade e menor curva de aprendizado em comparação a modelos como CUDA.

Recomendações para Melhorias

1. Utilizar `nowait` para evitar sincronizações desnecessárias se houver múltiplas regiões paralelas.
2. Explorar `reduction` caso haja agregações futuras (como somatórios globais).
3. Avaliar uso de `firstprivate` e `private` para controle fino de variáveis em diferentes escopos.
4. Implementar versão com `#pragma omp loop` separadamente para avaliação comparativa.
5. Comparar com implementações em CUDA e SYCL para análise de produtividade e performance.

Apêndice – Comparativo entre Modelos de Programação Paralela

Modelo	Descrição
CUDA	Principal linguagem da Nvidia para GPU. Baixo nível, alto desempenho, mas menos portável.
SYCL	Padrão aberto baseado em C++ moderno. Portável entre diferentes hardwares, promovido por Intel.
OpenAcc	Padrão aberto mais descritivo, com menos controle que CUDA, mas mais simples. Também compatível com GPUs Nvidia.
OpenMP	Ampliado para suportar offloading com pragmas `target`. Portável, com sintaxe simples. Fácil de integrar a códigos C/C++ existentes.

Principais cláusulas em OpenMP GPU:

- **`#pragma omp target`**: offload de região para dispositivo.
- **`map(to/from/alloc/release)`**: gerencia a movimentação de dados entre host e GPU.
- **`device(n)`**: especifica o dispositivo de execução.
- **`private`** / **`firstprivate`**: variáveis privadas por thread.
- **`nowait`**: execução sem bloqueio/sincronização.
- **`#pragma omp loop`**: controle fino sobre loops com `reduction`, `collapse`, `schedule`, etc.
