Relatório Tarefa 22: Simulação da Equação de Navier-Stokes com CUDA

Aluno: Cristovao Lacerda Cronje

---

1. Introdução

A tarefa propõe a implementação de uma simulação simplificada da equação de Navier-Stokes utilizando a linguagem CUDA para execução na GPU. O objetivo é explorar o paralelismo massivo da arquitetura CUDA e compará-lo com uma versão paralela tradicional baseada em OpenMP, executada na CPU. A equação utilizada modela um processo de difusão em um meio tridimensional, representando fenômenos como o transporte de calor ou a movimentação de fluidos em condições simplificadas.

---

2. Objetivos

- Implementar um kernel CUDA para simular a equação de calor (forma simplificada da equação de Navier-Stokes);
- Realizar a simulação em um domínio tridimensional;
- Comparar o desempenho da versão CUDA com implementações equivalentes em CPU utilizando OpenMP com 32 e 64 threads;
- Analisar os ganhos de desempenho e discutir as diferenças entre as abordagens CPU vs. GPU.

---

3. Metodologia

3.1 Estrutura do Código

A função `main` é responsável por alocar e inicializar a matriz tridimensional que representa o domínio da simulação. A condição inicial consiste em um pico de calor centralizado no espaço, simulando uma perturbação localizada. Após a configuração inicial, os dados são repassados para a função `simulate_cuda`, que conduz toda a simulação na GPU.

Na `simulate_cuda`, duas matrizes (representadas como vetores 1D) são alocadas na memória global da GPU: `vold`, com os valores anteriores, e `vnew`, com os valores atualizados. Em cada um dos 2000 passos de tempo da simulação, o kernel `atualiza` é executado para aplicar um esquema de diferenças finitas em três dimensões, baseado em um stencil 3D de 7 pontos. Esse stencil considera os seis vizinhos diretos de cada célula (em x, y e z), permitindo simular a propagação do calor.

Após cada passo de tempo, os ponteiros `vold` e `vnew` são trocados, evitando a necessidade de cópias adicionais. Além disso, a cada iteração, o valor da célula central do domínio é registrado, e o

tempo de execução daquele passo é medido com precisão usando `cudaEventRecord`. Isso permite avaliar o desempenho e acompanhar a evolução da simulação ao longo do tempo.

3.2 Kernel CUDA

O kernel `atualiza` é executado por uma grade tridimensional de blocos CUDA. Cada bloco contém 8×8×8 threads, totalizando até 512 threads por bloco, aproveitando o alto grau de paralelismo das GPUs.

Cada thread é responsável por atualizar um ponto específico do domínio tridimensional. Para localizar a célula correta na matriz, as coordenadas (x, y, z) são calculadas dinamicamente com base na posição da thread dentro do bloco e na posição do bloco dentro da grade. Essas coordenadas tridimensionais são convertidas para um índice linear, usado para acessar corretamente os elementos nos vetores 1D `vold` e `vnew`.

Esse mapeamento eficiente entre coordenadas 3D e vetores lineares é essencial em aplicações CUDA, pois permite o uso da memória global da GPU com acesso coalescido, otimizando o desempenho. A operação realizada por cada thread aplica o stencil de 7 pontos, atualizando a célula com base nos valores dos seus vizinhos imediatos em cada direção.

O cálculo é baseado no operador laplaciano 3D:

```
laplacian = (vold[idx+1] + vold[idx-1] +
             vold[idx+nx] + vold[idx-nx] +
             vold[idx+nx*ny] + vold[idx-nx*ny] -
             6 * vold[idx]) / (dx * dx);
```

Esse valor é então usado para atualizar o ponto atual, considerando os parâmetros físicos de difusão (`nu`, `dt` e `dx`).

3.3 Configuração e Parâmetros

Os parâmetros utilizados foram:

```
#define NX 300
#define NY 300
#define NZ 300
#define NSTEPS 2000

const double nu = 0.1;
const double dt = 0.01;
const double dx = 1.0;
```

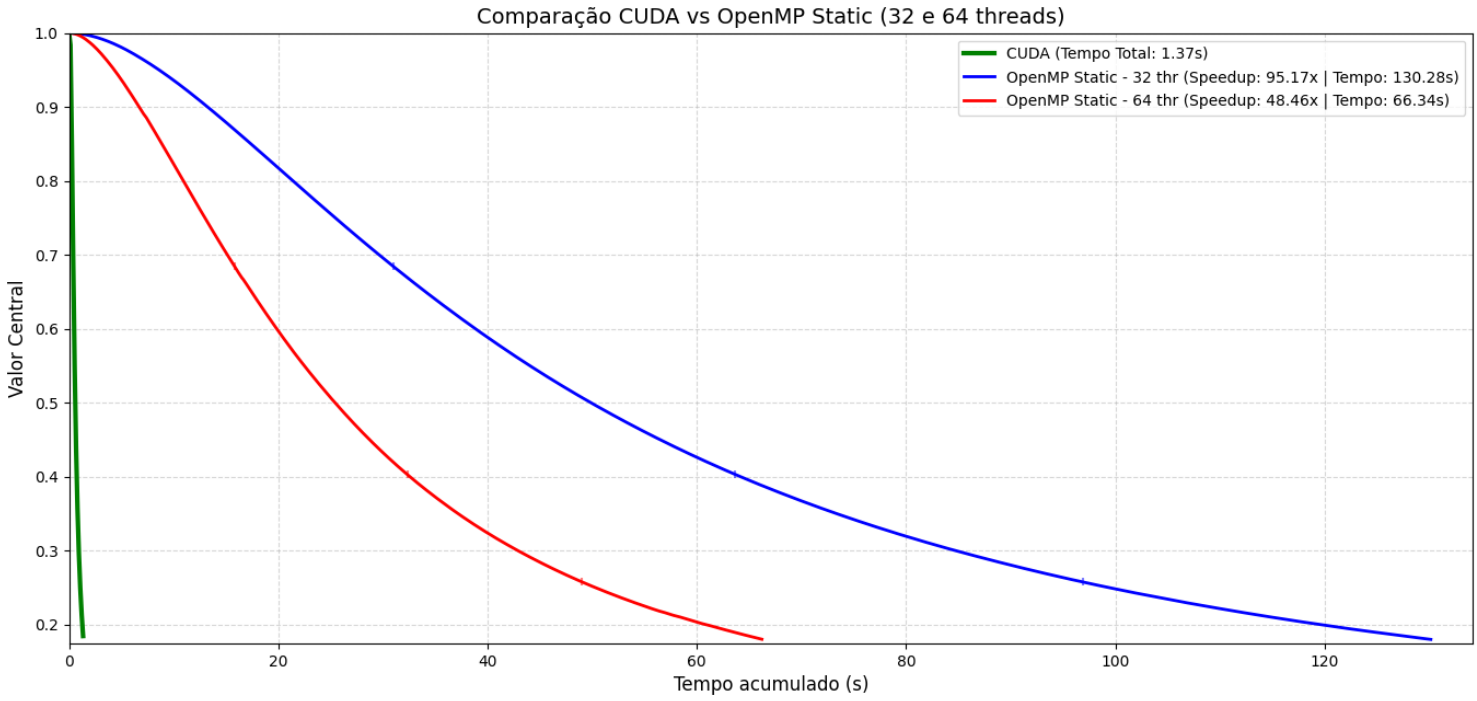
- O domínio tridimensional tem 300x300x300 pontos;
- A simulação executa 2000 passos de tempo;
- O valor de `alpha = nu * dt / (dx * dx)` representa o coeficiente de difusão.

---

4. Resultados e Análise

4.1 Resultados Obtidos

Abordagem	Threads	Tempo Total (s)
OpenMP	32 threads	130.28
OpenMP	64 threads	66.34
CUDA (GPU)	-	**1.37**



4.2 Explicação dos Resultados

O desempenho da versão CUDA é significativamente superior à das versões em OpenMP. A GPU completou a simulação em apenas **1.37 segundos**, contra **66.34 segundos** na CPU com 64 threads (e ainda pior com 32 threads).

Isso se explica pelas diferenças arquiteturais entre CPU e GPU:

- **CPUs** têm poucos núcleos altamente otimizados para execução sequencial ou paralelismo moderado via threads;
- **GPUs**, por outro lado, possuem milhares de núcleos menores capazes de executar milhares de threads em paralelo, otimizadas para workloads de alto paralelismo de dados, como esta simulação.

Além disso, o código CUDA aproveita o paralelismo tridimensional de forma natural, mapeando cada thread diretamente a um ponto do domínio 3D. Já na versão com OpenMP, o paralelismo é implementado por meio de diretivas em loops aninhados (por exemplo, sobre os eixos x, y e z), o que pode introduzir sobrecarga adicional. Isso inclui tempo de criação e gerenciamento de threads, acesso não otimizado à memória (cache e RAM), e competição por largura de banda entre os núcleos da CPU — fatores que limitam a escalabilidade e o desempenho em arquiteturas multicore tradicionais.

4.3 Impacto da Paralelização

A melhoria de desempenho na GPU demonstra a vantagem do paralelismo massivo para operações fortemente regulares, como a aplicação repetitiva de um stencil 3D.

Mesmo com 64 threads, a CPU não consegue se aproximar do desempenho da GPU. A execução em CUDA apresenta **speedup superior a 48x** em relação à versão OpenMP com 64 threads.

Esse resultado reforça a importância de considerar a arquitetura adequada para problemas de computação científica intensiva, onde o uso de GPU pode reduzir drasticamente o tempo de execução.

---

5. Conclusão

A tarefa demonstrou de forma clara o potencial de aceleração proporcionado pela arquitetura CUDA em simulações numéricas tridimensionais. A GPU superou amplamente o desempenho da CPU, mesmo utilizando paralelismo com OpenMP.

Além de reforçar conceitos fundamentais de paralelismo em CPU e GPU, o experimento evidenciou que problemas com alta regularidade espacial e temporal — como simulações baseadas em stencil — são ideais para execução em arquiteturas de processamento massivamente paralelo, como as GPUs.