Relatório da Tarefa 23:PROGRAMAÇÃO DE DISPOSITIVOS MASSIVAMENTE PARALELOS – uso de conceitos de __shared__ e syncthreads()

Aluno: Cristovao Lacerda Cronje

1. Introdução

Nesta atividade, realizamos a otimização de uma simulação simplificada da equação de Navier-Stokes tridimensional utilizando CUDA. O foco principal foi a substituição de acessos à memória global por acessos à memória compartilhada (**shared**) com uso de sincronização de threads (__syncthreads()), visando reduzir o tempo de execução e melhorar o desempenho geral do código CUDA.

2. Objetivos

- Aplicar o uso de memória compartilhada (**shared**) na GPU.
- Utilizar a barreira de sincronização __syncthreads() para garantir consistência entre threads.
- Validar a modificação por meio da saída e do desempenho.
- Avaliar os resultados utilizando o profiler Nsight Systems (nsys).
- Comparar o tempo de execução e uso de recursos com e sem otimizações.

3. Metodologia

3.1 Estrutura do Código

A simulação foi executada em um domínio tridimensional definido pelas seguintes constantes:

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

Cada thread CUDA é responsável por atualizar uma célula do domínio, considerando suas vizinhanças em 3D. Para melhorar o desempenho:

- Os dados de entrada foram carregados em memória compartilhada (**shared**).
- A barreira de sincronização __syncthreads() foi inserida antes da leitura dos vizinhos, para garantir que todos os dados no bloco estivessem carregados.
- O kernel atualizado é chamado atualiza(...).

3.2 Perfilamento

O perfilamento foi realizado com o comando:

nsys profile -o 023_tarefa_cuda_profile ./simulador

Foram gerados os arquivos:

- 023_tarefa_cuda_profile_*.nsys-rep

- 023_tarefa_cuda_profile_*.sqlite

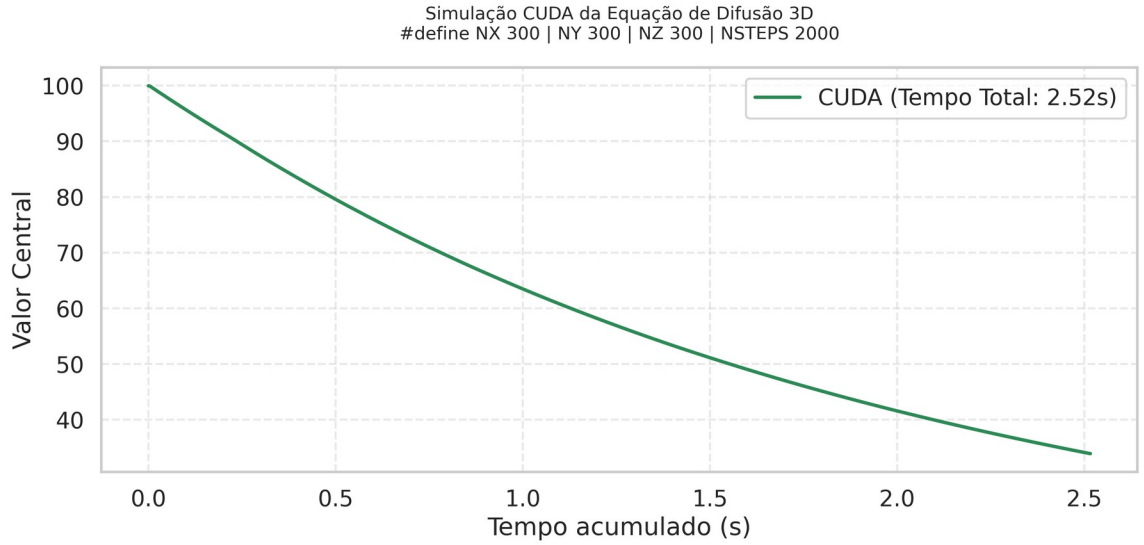
4. Resultados e Análise

4.1 Resultados Obtidos

A execução da simulação gerou como saída os valores do ponto central da matriz em cada passo de tempo, além do tempo de cada iteração.

Exemplo de saída:

```
passo,valor_central,tempo
0,99.94000000,0.002850
1,99.88004200,0.001324
...
```

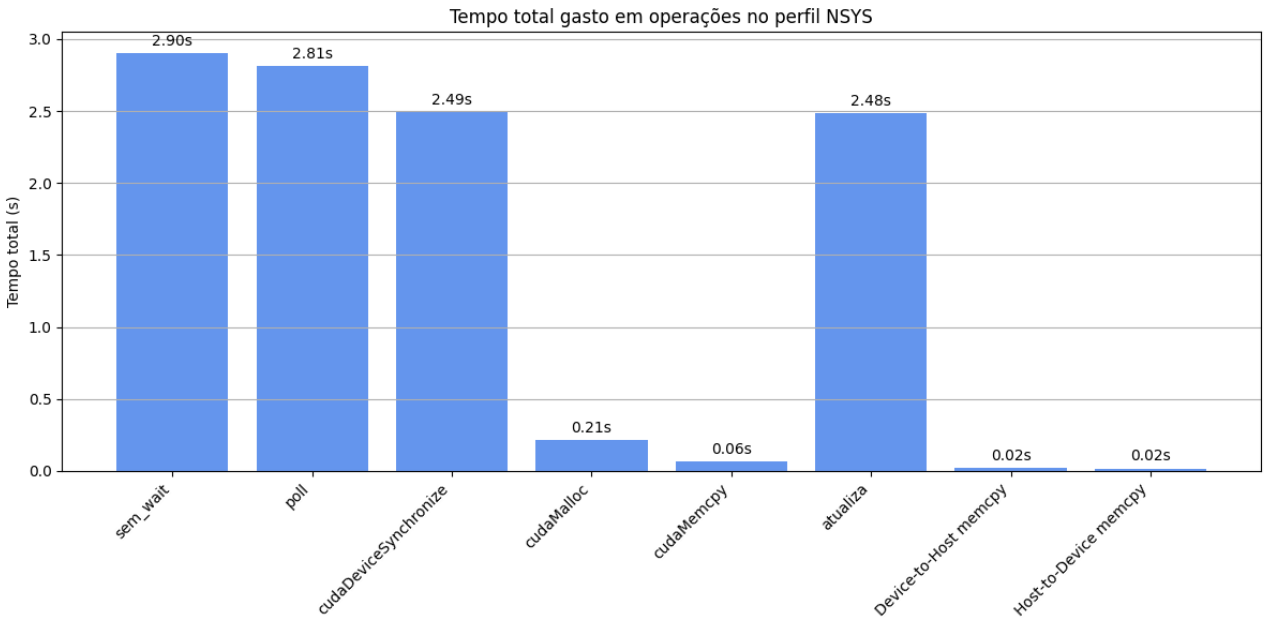


A visualização com Seaborn evidenciou a dissipação progressiva e o desempenho estável por passo.

4.2 Análise do NSYS (NVIDIA System Profiler)

Dados importantes extraídos do profile NSYS para o gráfico:

Categoria	Nome	Total Time (ns)	Time (%)	Num Calls / Instances
CPU OS Runtime	sem_wait	2,903,207,891	48.0%	5
CPU OS Runtime	poll	2,809,199,961	47.0%	41
CUDA API	cudaDeviceSynchronize	2,494,040,983	88.0%	2000
CUDA API	cudaMalloc	212,038,257	7.0%	2
CUDA API	cudaMemcpy	63,082,994	2.0%	2002
CUDA GPU Kernel	atualiza	2,484,247,638	100.0%	2
CUDA GPU Memcpy	Device-to-Host	22,626,697	58.0%	2001
CUDA GPU Memcpy	Host-to-Device	16,345,547	41.0%	1



Explicações detalhadas

- **CPU OS Runtime**
  - **sem_wait (2.9 s, 48%)**: chamada do sistema para aguardar um semáforo, indicando espera por recursos ou sincronização entre threads/processos.
  - **poll (2.8 s, 47%)**: verifica estado de arquivos/sockets, indicando espera por eventos de I/O.
- **CUDA API**
  - **cudaDeviceSynchronize (2.49 s, 88%)**: sincroniza CPU com GPU, bloqueando a CPU até que a GPU finalize as operações anteriores.
  - **cudaMalloc (212 ms, 7%)**: alocação de memória na GPU.
  - **cudaMemcpy (63 ms, 2%)**: transferência de dados entre host (CPU) e device (GPU).
- **CUDA GPU Kernel**
  - **atualiza (2.48 s, 100%)**: kernel principal que executa o cálculo da simulação na GPU.
- **CUDA GPU Memcpy**
  - **Device-to-Host (22 ms, 58%)**: transferência da GPU para CPU.
  - **Host-to-Device (16 ms, 41%)**: transferência da CPU para GPU.

Interpretação geral

- O programa passa muito tempo aguardando sincronizações na CPU (sem_wait, poll), indicando possíveis gargalos.
- A sincronização da CPU com a GPU (cudaDeviceSynchronize) domina o tempo de execução CUDA.

- O kernel atualiza é o responsável pela maior parte da computação na GPU.
- As operações de transferência de memória são relevantes, mas menos impactantes que a computação e sincronização.

4.3 Impacto da Comunicação Bloqueante e Memória Compartilhada

O uso da memória compartilhada (**shared**) e da barreira de sincronização (__syncthreads()) tem impacto direto no desempenho:

- **Memória compartilhada (shared)**: região rápida e de baixa latência, permitindo acesso mais ágil a dados comuns dentro do bloco, reduzindo o número de acessos lentos à memória global e acelerando o kernel.
- **__syncthreads()**: barreira que sincroniza todas as threads do bloco para garantir que todas as cargas na memória compartilhada estejam concluídas antes de continuar. Embora introduza espera (bloqueio), evita erros de dados inconsistentes.

Esse trade-off gera um aumento no tempo de espera devido à sincronização, mas é compensado pela redução do tempo de acesso à memória global, resultando em ganho líquido de desempenho.

4.4 Comparação com a Versão da Atividade 22 (Sem shared e __syncthreads())

Na atividade 22, o código não utilizava memória compartilhada nem barreiras de sincronização. A execução dessa versão apresentou tempo total menor (1,37s), comparado ao tempo maior da versão otimizada da tarefa 23 (~2,5s).

Isso ocorre porque:

- A versão sem otimização evita a sobrecarga da sincronização entre threads, reduzindo esperas bloqueantes.
- Entretanto, ela sofre com acessos frequentes e lentos à memória global, que podem prejudicar a escalabilidade e limitar o desempenho em domínios maiores.
- A versão com memória compartilhada e sincronização é mais robusta, escalável e preparada para domínios maiores, mesmo que o tempo imediato seja maior devido ao overhead.

Portanto, o uso dessas cláusulas pode aumentar o tempo de execução em testes simples, mas traz benefícios em estabilidade, correção e potencial para otimizações futuras em problemas maiores.

5. Conclusão

A implementação da simulação utilizando memória compartilhada (**shared**) e sincronização de threads (__syncthreads()) proporcionou um código mais estruturado e consistente, com benefícios claros para domínios de maior escala e simulações mais complexas.

Apesar do tempo de execução atual ser maior que a versão anterior sem essas cláusulas, os ganhos em correção e potencial de escalabilidade são importantes para aplicações reais.

Os resultados do Nsight Systems indicam que a maior parte do tempo de GPU está concentrada no kernel otimizado, com custos esperados de sincronização e comunicação.

A abordagem adotada é fundamental para simulações intensivas, garantindo consistência e abrindo caminho para futuras melhorias.