Universidade Federal de São Carlos
Departamento de Computação

# Programação Paralela

## ARQUITETURAS E ESTRATÉGIAS DE ACELERAÇÃO

Aluno: **Thiago Martins**

**Prof. Ricardo Menotti**

**Prof. Hélio Crestana Guardia**

São Carlos, 8 de maio de 2023

# 1 Descrição do problema

O algoritmo *Nbody* é utilizado amplamente em simulações em diversas áreas: Analise de Fluidos, Colisão de Corpos e simulações aerodinâmicas são alguns dos exemplos de seu potencial de uso. A gama de aplicações deste tipo de algoritmo elenca uma grande importância para seu uso, bem como as diversas utilizações que este algoritmo contribui. Para o caso em questão, é realizada a análise de $n$ partículas independentes entre si, com o objetivo de atualizar os seguinte dados: posição $(x, y, z)$ e velocidade $(vx, vy, vz)$. Com isso, como a modelagem não possui dependências entre os dados, esta implementação pode ser englobada em granularidade fina.

# 2 Estratégia de paralelização

## 2.1 Estratégia Utilizando openMP

Como estratégia inicial, após a devida análise de dependências entre os dados, foi tomada a decisão da implementação na linguagem openMP[1]. Esta decisão inicial no que se refere ao desenvolvimento em openMP se deve pelo fato haver maior facilidade para testes de dependências de paralelismo, bem como um meio para obter a melhor configuração para futuro paralelismo com uso de dispositivo de aceleração. Por meio de diretivas de compilação, a API openMP flexibiliza a criação de threads (ou contextos) na implementação em questão. As alterações realizada na implementação podem ser observadas nas imagens 1 e 2.

Nas implementações apresentadas em 1 2, é notavel o uso das diretivas para openMP voltado para CPU. Para ambos os casos, o uso do pragma utilizado é *parallel for*, no qual é criado um contexto em que cada thread tem sua execução desvinculada em relação às outras. Além disso, a criação de threads permite a execução em paralelo de i-ésimos valores individualmente - levando em conta limitações de número máximos de threads. Para o caso 2, é utilizada a diretiva *shared(bodies)*, cujo objetivo é permitir

[1] API e conjunto de diretivas de compilação que permite o desenvolvimento de aplicações com processamento paralelo para utilizar todo o potencial de processadores multi-core

```c
void body_force(Body *p, float dt, int n) {
  //Big-O --> n^2
  for (int i = 0; i < n; ++i) {
    float fx = 0.0f;
    float fy = 0.0f;
    float fz = 0.0f;

    for (int j = 0; j < n; j++) {
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float dz = p[j].z - p[i].z;
      float sqrd_dist = dx*dx + dy*dy + dz*dz + SOFTENING;
      float inv_dist = 1 / sqrt(sqrd_dist);
      float inv_dist3 = inv_dist * inv_dist * inv_dist;
```

```c
void body_force(Body *p, float dt, int n) {
  //Big-O --> n^2
  int i,j;
  #pragma omp parallel for private(i,j)
  for (int i = 0; i < n; ++i) {
    float fx = 0.0f;
    float fy = 0.0f;
    float fz = 0.0f;

    for (int j = 0; j < n; j++) {
      float dx = p[j].x - p[i].x;
      float dy = p[j].y - p[i].y;
      float dz = p[j].z - p[i].z;
      float sqrd_dist = dx*dx + dy*dy + dz*dz + SOFTENING;
      float inv_dist = 1 / sqrt(sqrd_dist);
      float inv_dist3 = inv_dist * inv_dist * inv_dist;
```

Figura 1: Função *Body-force*: código inicial à esquerda, openMP à direita

```c
int iter, i;
for (iter = 0; iter < N_ITER; iter++) {
  body_force(bodies, DT, nbodies);
  for (i = 0; i < nbodies; i++) {
    bodies[i].x += bodies[i].vx * DT;
    bodies[i].y += bodies[i].vy * DT;
    bodies[i].z += bodies[i].vz * DT;
  }
}
```

```c
int iter, i;
#pragma omp parallel for private(iter) shared(bodies)
for (iter = 0; iter < N_ITER; iter++) {
  body_force(bodies, DT, nbodies);
  for (i = 0; i < nbodies; i++) {
    bodies[i].x += bodies[i].vx * DT;
    bodies[i].y += bodies[i].vy * DT;
    bodies[i].z += bodies[i].vz * DT;
  }
}

write_dataset(nbodies, bodies, file_cpu);
```

Figura 2: loop principal: código inicial à esquerda, openMP à direita

que o mesmo dado seja compartilhado para todas as threads. Ou seja, cada partícula é calculada independemente e paralelamente ao longo das iterações, de acordo com o fator limitante do total de threads permitidas simultaneamente.

Tal passo permitiu observar que o maior custo está relacionado com a quantidade de partículas, dado que foi obtido com esta implementação uma aceleração próximo de 3.5 vezes em relação ao inicial, considerando casos de números de corpos relativamente grandes (nbodies $>$ 30.000). Com isso, foi possível notar alto desempenho na resolução do problema encontrado, conforme esperado.

## 2.2 Estratégia Utilizando CUDA

### 2.2.1 Implementação Inicial

Nesta etapa, com os conceitos abordados na implementação de openMP, o desenvolvimento dos kernels de offload para a GPU elencou conceitos explorados na linguagem de openMP. Entretanto, para plataforma com aceleradores, deve-se avaliar cada caso,

pois o tamanho total de dados pode influenciar no tempo total de execução(transmissão entre Host e Device). Por conta disso, é crucial observar tempos como: transmissão de memória Host para Device, tal como Device para host, pois podem ser vistos como um gargalo de operação. Para isso, foi conclusivo quando apresentado acima de 1000 elementos, passa a ser viável a transmissão de dados para o Acelerador (neste caso, uma placa de Vídeo). Com isso, apresenta-se os kernels desenvolvidos para este problema nas imagens 3:

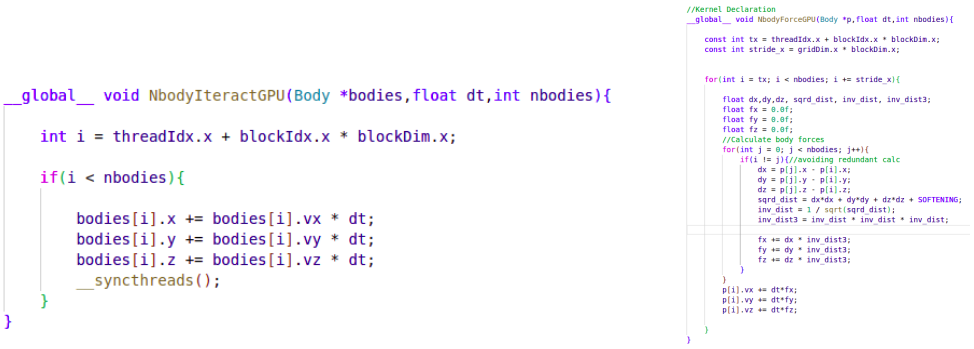


Figura 3: Função *CUDA*: Declaração de Kernels

Apresentado inicialmente os kernels, o resultante do encapsulado é obtido por meio deste 4


Figura 4: Declaração de Encapsulador

O resultado de transmissões de kernel apresentaram maior demanda justamente no cálculo de interação entre corpos, lembrando que o rendimento é efetivo a partir de 1.000 elementos. É observável o tempo de execução para cada etapa , levando em conta nbodies=20.000

Com isso, realiza-se o perfilamento com ferramentas específicas tal como *Nsight*

| Estágio | Tempo(ms) |
|---|---|
| Cópia p/ Device | 00.209 |
| Execução | 29.213 |
| Cópia p/ Host | 00.318 |

Tabela 1: Exemplo de Tempo de Execução

*Systems*, observado na imagem a 5. Esta ferramenta tem por objetivo analisar eventuais gargalos de execução de algoritmos, tal como avaliar possibilidades de uso para melhorias de código:

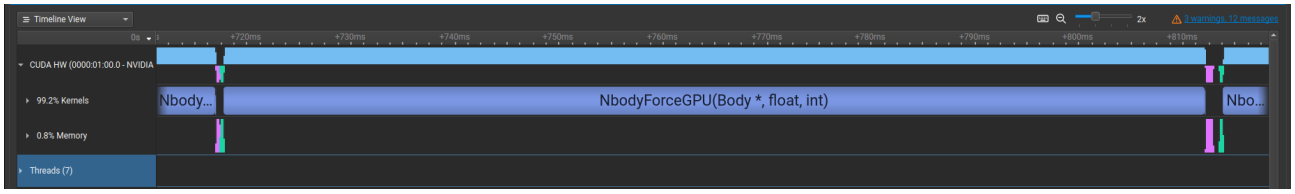


Figura 5: Apresentação Kernel

Como pode se notar, tem-se como maior gargalo a execução (em uma única stream) da função NbodyForceGPU. No tópico a seguir será abordado como extrair melhores resultados deste kernel 5.

### 2.2.2 Incluindo Streams

Por conta da independencia de dados presentes no caso em questão, uma possibilidade de estrair resultados é por meio do uso de *Streams*, cujo intuito é criar "filas"de execução contendo certas políticas de execução que permitem a execução assíncrona de Leitura, execução e Escrita no resultado esperado para comparação. PAra isso, é necessario a manipulação de *streams*, que requer inicializações prévias para seu funcionamento. Isso pode ser notado na declaração de kernel na imagem6.

A imagem 6 apresenta a divisão em multiplos *streams* a execução do algoritmo *nbodies*. Isso permite uma maior redução no tempo de execução do algoritmo, permitindo uma execução assíncrona de múltiplos kernels e melhorando o algoritmo. Isso permitiu a melhoria do resultado apresentado na tabela 2:

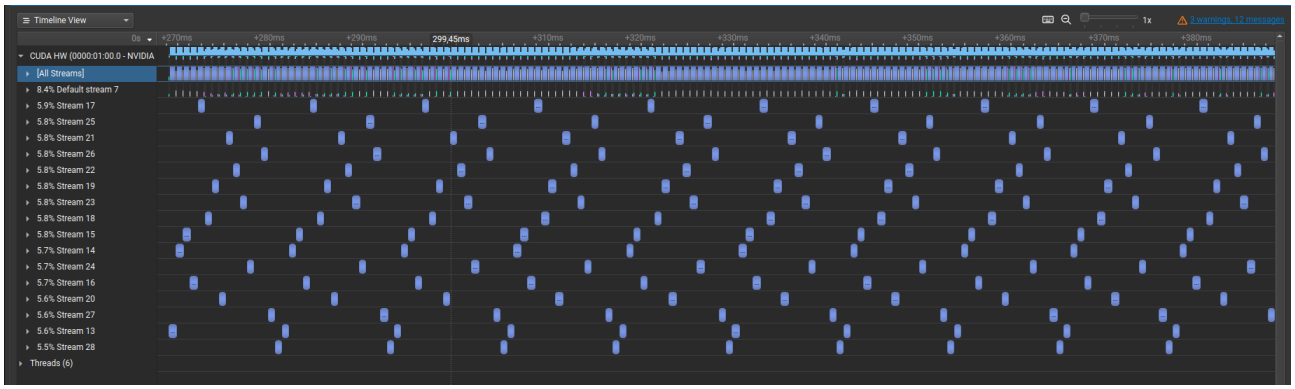


Figura 6: Declaração de Kernel com uso de streams

| Nro. Particulas | Exec. Stream Única(ms) | Exec. Multiplo Stream(ms) |
|---|---|---|
| 1000 | 220 | 180 |
| 1500 | 180 | 190 |
| 2000 | 180 | 190 |
| 2500 | 190 | 180 |
| 3000 | 190 | 190 |
| 3500 | 200 | 200 |
| 4000 | 210 | 200 |
| 4500 | 200 | 200 |
| 7000 | 230 | 230 |
| 10000 | 305 | 277 |
| 20000 | 510 | 387 |
| 25000 | 670 | 410 |
| 50000 | 1610 | 620 |

Tabela 2: Tempo médio de Execução - comparativo

## 3 Análise da escalabilidade: esperada e obtida

A análise da tabela de tempo médio de execução apresentada revela que o sistema apresentou uma boa escalabilidade, tanto na execução com stream única quanto na execução com múltiplos streams. Foi possível observar que, para a maioria dos valores de entrada, a execução com múltiplos streams apresentou uma melhora significativa em relação à execução com stream única, indicando que o sistema tem uma boa capacidade de aumentar o desempenho conforme aumenta-se a carga de trabalho, levando-se em consideração uma execução em GPU. Portanto, a escalabilidade esperada e obtida foi excelente, com a implementação em CUDA proporcionando um desempenho superior em relação às outras plataformas de desenvolvimento, mostrando que o tempo investido no desenvolvimento foi recompensado pelo ganho em eficiência do sistema. Isso destaca

a importância da escolha da plataforma de desenvolvimento adequada para garantir uma boa escalabilidade do sistema. Isso demonstra a efetividade para este algoritmo, bem como uma diferente estratégia para abordar o problema em questão.

## 4 Discussão sobre a eficiência da solução

Portanto há de se concluir que a paralisação é uma maneira efetiva de se obter resultados em relação ao tempo de execução. Pode-se observar que é intrinsecamente relacionado o tipo de plataforma com o dispositivo acelerador utilizado: openMP voltado para CPU, CUDA voltado para GPU. Com isso, o processamento nessas diferentes plataformas tem resultados completamente distintos. Foi observado que o resultado obtido das implementações em CUDA são satisfatórios, dado que a granularidade do algoritmo contribuiu para o sistema em questão. O gráfico a seguir destaca os desempenhos comparativos em relação as plataformas, revelando o incrível potencial do uso de programação Paralela. A partir do gráfico é conclusivo que o resultado para GPU's do que se refere ao tempo de execução, são quase insignificantes em relação as operações feitas em CPU.

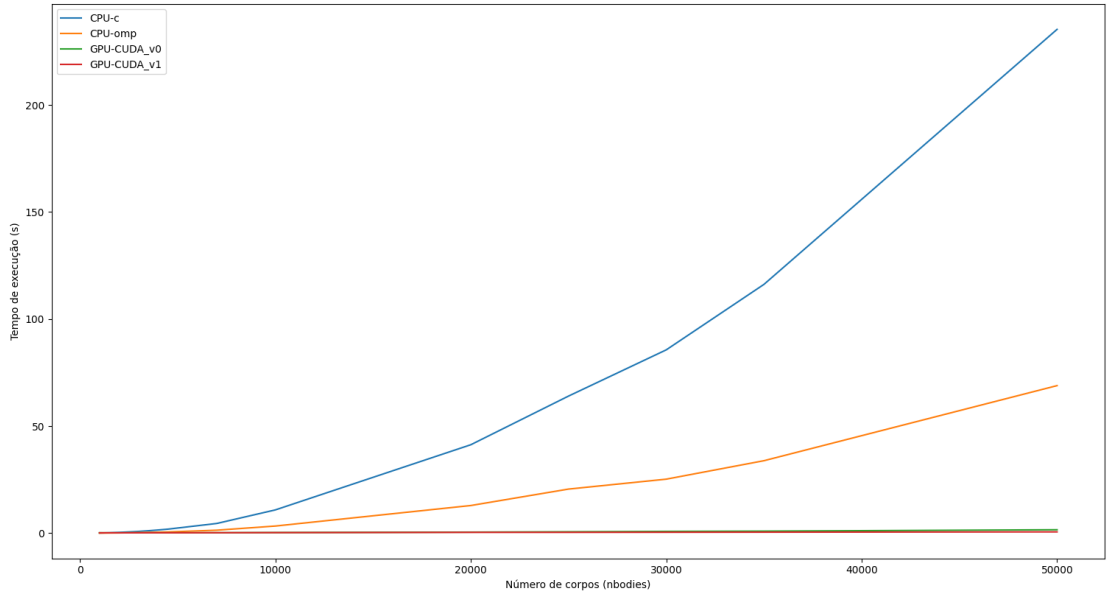


Figura 7: Comparativo entre plataformas

# 5 Conclusões

Foi conclusivo que durante o experimento, o algoritmo Nbody revela uma capacidade de execução preferencialmente em gpus seu rendimento é superiormente melhor(levando em conta um número suficiente alto de particulas). Isso permite concluir que a aceleração foi satisfatoriamente concluída levando em conta por aceleradores, que neste caso é a GPU. Há de concluir que este trabalho motivou o uso prático e teórico, com o objetivo de melhorias intrínsecas de execução. Isso demonstra que a efetividade para este algoritmo foi positiva levando em conta um número considerável de carga para execução no acelerador.