Exercício de Progração

Disciplina Organização e Arquitetura de Computador II

November 25, 2025

Objetivo da Atividade

O objetivo desta atividade é comparar o desempenho de diferentes estratégias de paralelização aplicadas ao mesmo algoritmo: a convolução 2D utilizada em processamento de imagens.
Cada grupo, composto por 04 alunos, deverá implementar **quatro versões**:

1. Versão sequencial (sem paralelismo)
2. Versão com threads explícitas (pthread ou std::thread)
3. Versão com paralelismo implícito (OpenMP)
4. Versão executada em GPU (CUDA ou OpenMP target)

Imagem e Kernel Utilizados

A imagem base fornecida é um arquivo 20x20, mas deve ser redimensionada para comparação em diferentes escalas.

Resoluções obrigatórias:

- 512×512
- 1024×1024
- 4096×4096

Kernel 3x3 utilizado (blur):

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Todos os grupos devem usar o mesmo kernel.

Versão 1: Implementação Sequencial

A primeira versão deve ser desenvolvida sem uso de qualquer técnica de paralelismo.

- Implementar convolução 2D com loops aninhados.
- Não usar otimizações, paralelismo ou vetorizações.
- Esta versão serve como **baseline** para calcular speedup.

Versão 2: Threads Explícitas

Implementar o paralelismo manual utilizando `pthread` ou `std::thread`.

- Dividir manualmente a imagem entre threads.
- Cada thread deve processar uma região (linhas ou blocos).
- Número de threads deve ser informável por parâmetro.
- Garantir acesso seguro à matriz de saída.

Versão 3: Paralelismo Implícito com OpenMP

Nesta etapa, paralelize o algoritmo usando `#pragma omp parallel for`. Testar diferentes estratégias:

- `schedule(static)`
- `schedule(dynamic)`
- `collapse(2)`

Comparar o impacto de cada uma no tempo de execução.

Versão 4: Execução em GPU

A versão em GPU pode ser implementada de duas formas:

a) CUDA

- Usar grids 16×16 ou 32×32.
- Implementar versões com e sem shared memory.

b) OpenMP Target Offloading

- Usar diretiva `target teams distribute parallel for`.
- Considerar custo de transferência Host–Device.

Medição de Desempenho

Para cada versão e tamanho de imagem, medir:

- Tempo total de execução
- Média de 10 execuções
- Desvio padrão

Importante: medir apenas a convolução, não a leitura da imagem.

Speedup e Eficiência

Speedup:

$$S = \frac{T_{\text{seq}}}{T_{\text{paralelo}}}$$

Eficiência (para CPU):

$$E = \frac{S}{N_{\text{threads}}}$$

As métricas devem ser analisadas em função de:

- Tamanho da imagem
- Número de threads
- Tipo de paralelismo

Comparações Obrigatórias

O relatório deve comparar:

- Threads explícitas vs OpenMP
- CPU vs GPU
- Efeito do escalonamento no OpenMP
- Impacto do tamanho da imagem na escalabilidade

Além disso, discutir:

- Overhead de criação de threads
- Afinidade de cache
- Custo de transferência CPU–GPU

Entrega do Trabalho

O grupo deverá entregar:

- Data de entrega: 08/12
- Código-fonte das quatro versões
- Gráficos:
  - Speedup $\times$ Tamanho da imagem
  - Speedup $\times$ Nº de threads (exceto GPU)
  - Tempo total $\times$ Técnica
- Relatório contendo:
  - Metodologia
  - Resultados experimentais
  - Discussões
  - Conclusões sobre a melhor abordagem

Conclusões Esperadas

O grupo deverá responder:

- Qual técnica foi mais eficiente?
- A GPU superou a CPU? Em quais tamanhos?
- OpenMP apresenta desempenho similar a threads manuais?
- Onde o paralelismo não trouxe ganho?
- Qual abordagem oferece melhor custo-benefício?

Descrição das abordagens

Descrição das abordagens

Nesta seção, são descritas as técnicas utilizadas: sequencial, threads explícitas, OpenMP e GPU.

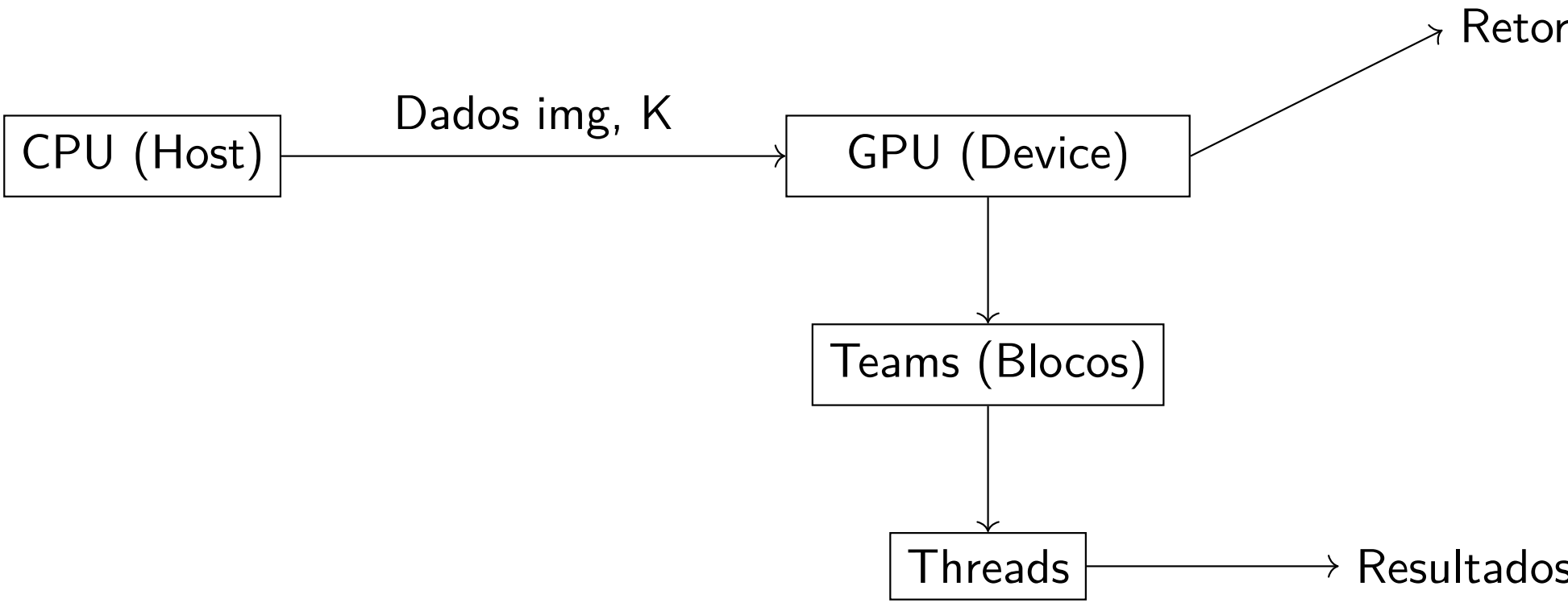
OpenMP Target Offloading (GPU)

OpenMP Target Offloading permite executar regiões do código em aceleradores (GPU) diretamente usando diretivas OpenMP.

- **Diretiva #pragma omp target**: marca a região de código para o dispositivo.
- **Mapeamento de memória**: variáveis precisam ser copiadas entre host e device usando `map(to/from/tofrom)`.
- **Teams e Threads**: `teams` cria grupos de threads, `parallel for` distribui o trabalho.

```
#pragma omp target teams distribute parallel for \
    map(to: img[0:H*W], K[0:9]) \
    map(from: out[0:H*W]) collapse(2)
for (int i = 0; i < H; i++)
    for (int j = 0; j < W; j++)
        out[i*W+j] = ...; // operação no device
```

Fluxo de Dados – OpenMP Target Offloading



Programação CUDA

Programação CUDA

CUDA é um modelo de programação paralela desenvolvido pela NVIDIA que permite executar kernels diretamente na GPU.

O que é CUDA?

CUDA (Compute Unified Device Architecture) é uma arquitetura e modelo de programação paralela para GPUs NVIDIA.

Conceitos principais:

- **Host e Device**: CPU e GPU.
- **Threads, Blocos, Grids**: organização hierárquica.
- **Kernels**: funções executadas paralelamente na GPU.

Memória em CUDA

CUDA oferece diferentes tipos de memória:

- **Global**: Alta latência, acessível por todas as threads e pelo host.
- **Shared**: Muito rápida, compartilhada entre threads do mesmo bloco.
- **Local**: Privada de cada thread.
- **Constant**: Somente leitura e cacheada.

Hierarquia de Memória CUDA

Global Memory
Alta latência

Shared Memory
Rápida

Local Memory
Por thread

Exemplo de Kernel CUDA

```
__global__ void add(int *a, int *b, int *c, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < N)
        c[idx] = a[idx] + b[idx];
}
```

Explicação: Cada thread soma um elemento dos arrays.