Paralelizando o código

Guia introdutório

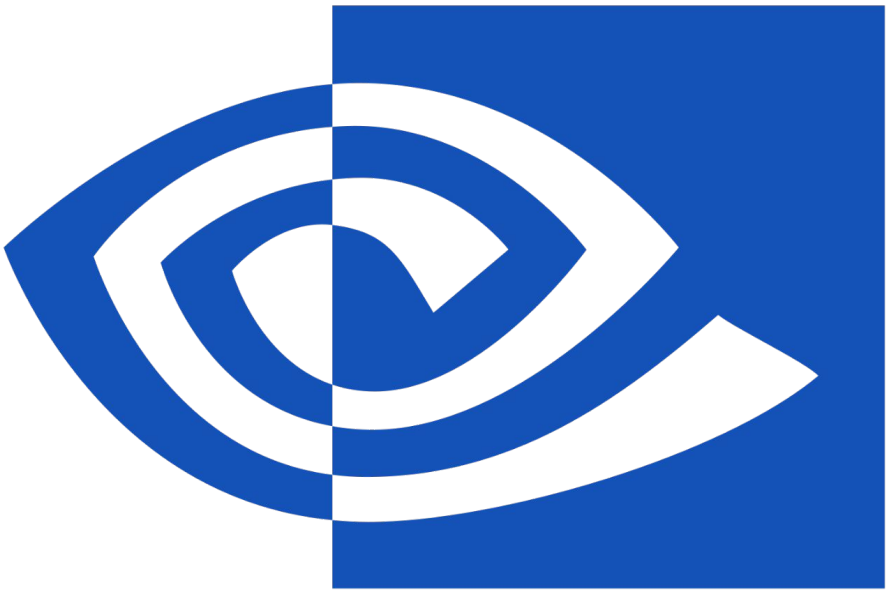
OpenMp e CUDA

Jansen Alves de Andrade

Paralelismo na Computação

Acelerando o Processamento

O **paralelismo na computação** é uma técnica que permite **executar múltiplas tarefas simultaneamente**, aproveitando o poder de processadores com vários núcleos (CPUs) ou unidades gráficas (GPUs). Em vez de processar uma instrução por vez, o sistema divide o problema em partes menores, executadas ao mesmo tempo. Essa estratégia é essencial em áreas como **inteligência artificial, simulações, análise de dados e gráficos 3D**, onde o volume de processamento é muito alto.



01

OpenMP: Paralelismo Simplificado em CPUs

O **OpenMP (Open Multi-Processing)** é uma API amplamente utilizada em linguagens como **C, C++ e Fortran** para **implementar paralelismo em processadores multicore**. Ele usa **diretivas de compilador** que indicam quais partes do código devem ser executadas em paralelo, sem que o programador precise gerenciar manualmente as threads.

OpenMP

Paralelismo Simplificado em CPUs

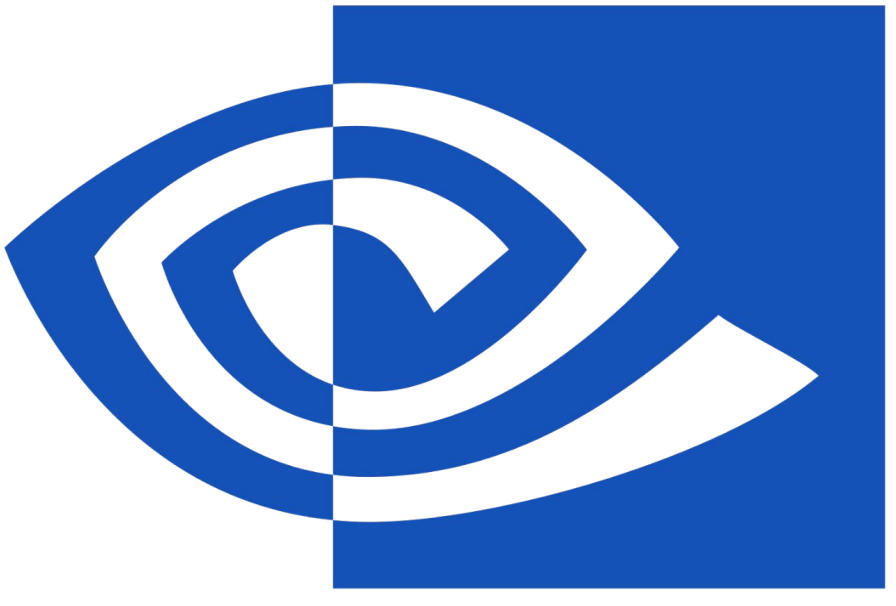
Principais diretivas do OpenMP:
#pragma omp parallel → Cria uma região paralela, onde várias threads executam o mesmo trecho de código.

#pragma omp for → Divide as iterações de um laço entre as threads.

#pragma omp sections → Permite executar diferentes blocos de código em paralelo.

#pragma omp critical → Garante que apenas uma thread execute um bloco de código por vez, evitando conflitos.

reduction → Combina resultados parciais (como somas) de forma segura.



OpenMP

Exemplos práticos

```c
#include <omp.h>
#include <stdio.h>

int main() {
    int n = 1000000;
    double soma = 0.0;
    double vetor[n];

    for (int i = 0; i < n; i++)
        vetor[i] = i * 0.5;

    #pragma omp parallel for reduction(+:soma)
    for (int i = 0; i < n; i++)
        soma += vetor[i];

    printf("Soma total = %.2f\n", soma);
    return 0;
}
```

#Nesse exemplo, cada thread soma uma parte do vetor, e o OpenMP combina os resultados automaticamente.
#O resultado é o mesmo de uma execução sequencial, mas com muito mais rapidez.

02

CUDA: Paralelismo Massivo em GPUs

O CUDA (Compute Unified Device Architecture), desenvolvido pela NVIDIA, é uma plataforma que permite programar GPUs para computação geral. Diferente das CPUs, que têm poucos núcleos potentes, as GPUs possuem milhares de núcleos menores, ideais para executar a mesma operação em muitos dados simultaneamente.

CUDA

Paralelismo Massivo em GPUs

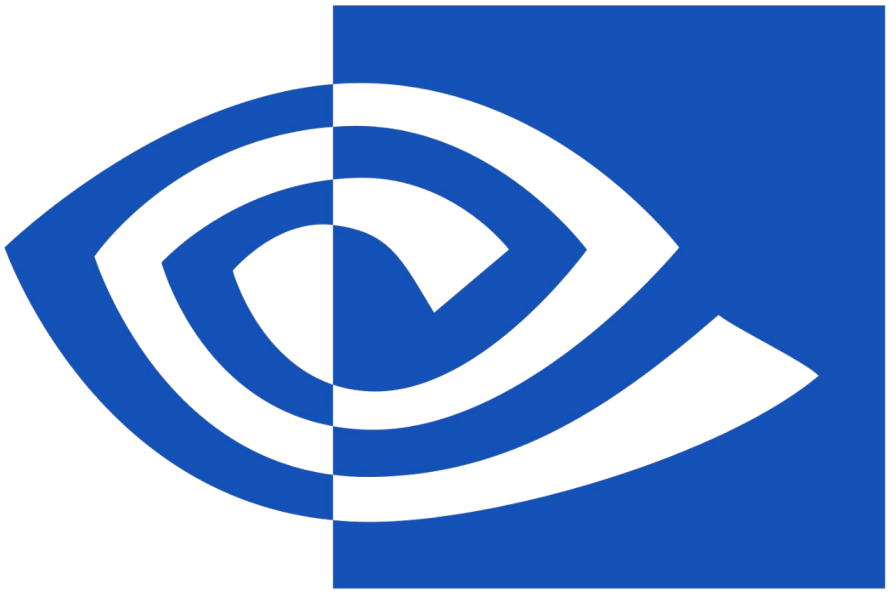
Conceitos fundamentais
Kernel → Função executada em paralelo por milhares de threads na GPU.

Thread, Block e Grid → Organização hierárquica das execuções.

Threads formam blocks, e blocks formam grids.

Memória global e compartilhada → Tipos de memória com diferentes velocidades e escopos de acesso.

cudaMemcpy() → Função usada para transferir dados entre CPU (host) e GPU (device).



CUDA

Exemplos práticos

```c
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void multiplicaVetores(float *a, float *b, float *resultado, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        resultado[i] = a[i] * b[i];
}

int main() {
    int n = 100000;
    size_t tamanho = n * sizeof(float);
    float *a, *b, *resultado, *d_a, *d_b, *d_resultado;

    a = (float*) malloc(tamanho);
    b = (float*) malloc(tamanho);
    resultado = (float*) malloc(tamanho);

    for (int i = 0; i < n; i++) {
        a[i] = i * 0.5;
        b[i] = i * 0.25;
    }

    cudaMalloc(&d_a, tamanho);
    cudaMalloc(&d_b, tamanho);
    cudaMalloc(&d_resultado, tamanho);

    cudaMemcpy(d_a, a, tamanho, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, tamanho, cudaMemcpyHostToDevice);

    int threads = 256;
    int blocos = (n + threads - 1) / threads;

    multiplicaVetores<<<blocos, threads>>>(d_a, d_b, d_resultado, n);

    cudaMemcpy(resultado, d_resultado, tamanho, cudaMemcpyDeviceToHost);

    printf("Primeiro resultado: %.2f\n", resultado[0]);

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_resultado);
    free(a); free(b); free(resultado);
    return 0;
}
```

03

Conclusão:
O Futuro é Paralelo

O OpenMP e o CUDA são ferramentas essenciais para quem busca alto desempenho computacional.
Enquanto o OpenMP é ideal para tarefas paralelas em CPUs, o CUDA domina o campo das execuções massivas em GPUs.
Em ambos os casos, o objetivo é o mesmo: dividir o trabalho para multiplicar a eficiência — o princípio fundamental do paralelismo moderno.

Agradecimentos

Este conteúdo foi desenvolvido com dedicação e cuidado, buscando sempre apresentar informações de forma clara, objetiva e útil para quem deseja aprender mais sobre paralelismo na computação.

Ressalto que o texto foi gerado com o apoio de Inteligência Artificial, sob minha coordenação e curadoria, garantindo a qualidade, organização e alinhamento do conteúdo com o propósito deste projeto.

Muito obrigado pela leitura.