Modello di Programmazione Cuda

Slides

Indice

Introduzione al Modello di Programmazione CUDA

L'ecosistema CUDA è organizzato in una struttura stratificata che bilancia semplicità d'uso e controllo hardware per ottimizzare le prestazioni. Include:

- **Applicazioni**: programmi paralleli su GPU.
- **Modello di programmazione**: astrazioni con thread, blocchi e griglie.
- **Compilatore/Librerie**: traduzione del codice in istruzioni GPU.
- **Sistema operativo**: gestione delle risorse tra applicazioni.
- **Architettura**: hardware NVIDIA.

Il **modello di programmazione CUDA** fornisce le regole per sviluppare applicazioni parallele, includendo:

1. **Gerarchia di Thread**: thread organizzati in blocchi e griglie.
2. **Gerarchia di Memoria**: memoria globale, condivisa, locale, costante e texture.
3. **API CUDA**: gestione kernel, trasferimenti dati, ecc.

Il **programma CUDA** specifica:

- Suddivisione ed elaborazione dei dati.
- Accesso e sincronizzazione dei thread.
- Esecuzione parallela delle operazioni.

Livelli di Astrazione

Il calcolo parallelo si articola in tre livelli:

1. **Dominio**: decomposizione del problema in unità parallele (es. matrici).

2. **Logico**: organizzazione dei thread per calcoli efficienti.
3. **Hardware**: ottimizzazione per risorse GPU.

Esempio banale: nella moltiplicazione di matrici:

- **Dominio**: suddivisione delle matrici.
- **Logico**: assegnazione calcoli ai thread.
- **Hardware**: accesso efficiente alla memoria.

Thread CUDA

Un thread CUDA rappresenta un'èunità di calcolo eseguita su GPU. Ogni thread:

- Esegue una parte del kernel.
- Opera su dati specifici determinati da **threadIdx** e **blockIdx**.
- Ha un proprio stato con registri e memoria locale.

Confronto CPU vs GPU:

- **GPU**: parallelismo massivo con migliaia di thread.
- **CPU**: parallelismo limitato.

Struttura del Programma CUDA

Caratteristiche principali:

1. **Codice ibrido**: esecuzione su CPU (host) e GPU (device).
2. **Kernel CUDA**: sezioni parallele eseguite su GPU.
3. **Esecuzione asincrona**: CPU e GPU possono lavorare in parallelo.
4. **Gestione dei risultati**: trasferimento dati GPU → CPU per elaborazioni successive.

Flusso Tipico di Elaborazione CUDA

1. Inizializzazione delle variabili e allocazione memoria.
2. Trasferimento dati da host a device.
3. Esecuzione del kernel su GPU.
4. Trasferimento risultati da device a host.
5. Elaborazione finale su CPU.
6. Rilascio delle risorse.

Gestione della Memoria in CUDA

CUDA divide la memoria in **host (CPU)** e **device (GPU)**. La comunicazione avviene tramite il bus PCIe.

Tipologie di memoria:

- **Globale**: accessibile da tutti i thread, grande ma lenta.
- **Condivisa**: veloce, ma condivisa solo tra i thread di un blocco.

Funzioni principali:

- **cudaMalloc**: alloca memoria su GPU.

- **cudaMemcpy**: trasferisce dati tra host e device.
- **cudaFree**: libera memoria GPU.

Esempio:

```
int *d_data;
cudaMalloc(&d_data, size);
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
cudaFree(d_data);
```

Organizzazione dei Thread in CUDA

CUDA utilizza una struttura gerarchica:

1. **Grid**: array di blocchi, rappresenta la computazione globale.
2. **Block**: gruppo di thread che condividono memoria.
3. **Thread**: unità minima di calcolo.

Ogni thread è identificato da **threadIdx** e ogni blocco da **blockIdx**. La dimensione di blocchi e griglie si definisce con il tipo **dim3**.

Esempio semplice (slide 15): somma di array:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < N) C[idx] = A[idx] + B[idx];
```

Kernel CUDA

Un kernel CUDA è una funzione eseguita in parallelo su GPU. Sintassi:

```
kernel_name<<<gridSize, blockSize>>>(args);
```

- **gridSize**: dimensioni della griglia (numero di blocchi).
- **blockSize**: dimensioni dei blocchi (numero di thread).

Qualificatori principali:

- **global**: funzione kernel eseguita su GPU, chiamata da CPU.
- **device**: funzione eseguita su GPU, chiamata da GPU.
- **host**: funzione eseguita su CPU.

Tecniche di Mapping e Dimensionamento

Per associare i thread ai dati, si utilizzano formule come:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

Es.: In una somma di array, ogni thread si occupa di un elemento.

Analisi e Ottimizzazione delle Prestazioni

CUDA offre strumenti come **NVIDIA Nsight** per:

- Profilare le prestazioni dei kernel.
- Identificare colli di bottiglia (es. trasferimenti di memoria).

Applicazioni Pratiche

1. **Operazioni su matrici**: suddivisione in blocchi.
2. **Elaborazione immagini**: conversione da RGB a grayscale.
3. **Convoluzione 1D e 2D**.

Nota: Per visualizzazioni utili, fare riferimento alle slide 10 (struttura CUDA) e 15 (esempio somma array).