Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Simulazione stocastica su Graphics Processing Unit di modelli di proliferazione cellulare

Relatore: *Prof. Daniela Besozzi*

Co-relatore: *Dr. Simone Spolaor*

Relazione della prova finale di:

Eric Nisoli

Matricola 807147

Anno Accademico 2017-2018

Indice

Capitolo 1

Introduzione

Capitolo 2

Metodi

2.1 Architettura CUDA

CUDA è l'architettura di elaborazione in parallelo progettata e sviluppata da NVIDIA che sfrutta la potenza di calcolo delle GPU (Graphics Processing Units) per aumentare le prestazioni nell'ambito del software computing. L'elaborazione sta lentamente migrando verso il paradigma di *co-processing* su CPU e GPU il quale prevede che l'esecuzione della gran parte del carico computazionale venga demandata alla GPU, e i risultati presi nuovamente in carico dalla CPU. Questo nuovo tipo di architettura ha trovato immediato seguito nel settore della ricerca scientifica, dato che ha contribuito in particolare alla nascita e al miglioramento di software per la simulazione di fenomi fisici e biologici.

Specifiche Hardware Generalmente l'hardware può cambiare con l'avvento di nuove generazioni di GPU ma la struttura generale si basa sempre sul concetto di Streaming Multiprocessors (SMs) [1, p. 62]

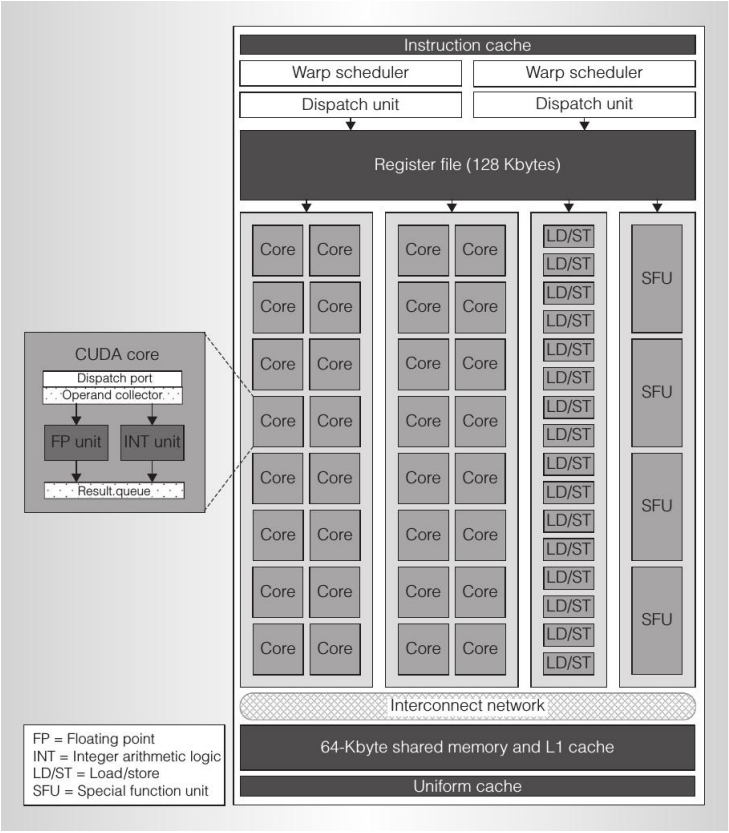

FIGURA 2.1: Streaming Multiprocessor dell'architettura Fermi [1, p. 63]

Astrazione Software Dato che la struttura fisica delle schede è in continuo mutamento, NVIDIA ha sviluppato delle API per dialogare con la GPU che sono indipendenti dall'architettura del device utilizzato, rendendo così possibile lo sviluppo di software portabile su molte schede che supportano CUDA.
Il modello astratto definisce tre tipologie di oggetti:

- **Thread**: singole unità di calcolo, eseguono il codice sorgente;
- **Thread block**: insieme logico di thread. I thread appartenenti allo stesso blocco hanno accesso ad un'area di memoria condivisa e accessibile solamente da loro (oltre alla memoria globale della GPU); inoltre è possibile ottenere un livello di sincronizzazione fra i thread del blocco;
- **Grid**: insieme logico di *Thread block*. Non è stata prevista un'area di memoria condivisa da tutta la griglia e fino ad ora non esiste una primitiva per sincronizzare i blocchi di una specifica griglia, è quindi necessario procedere alla sincronizzazione di tutta la GPU.

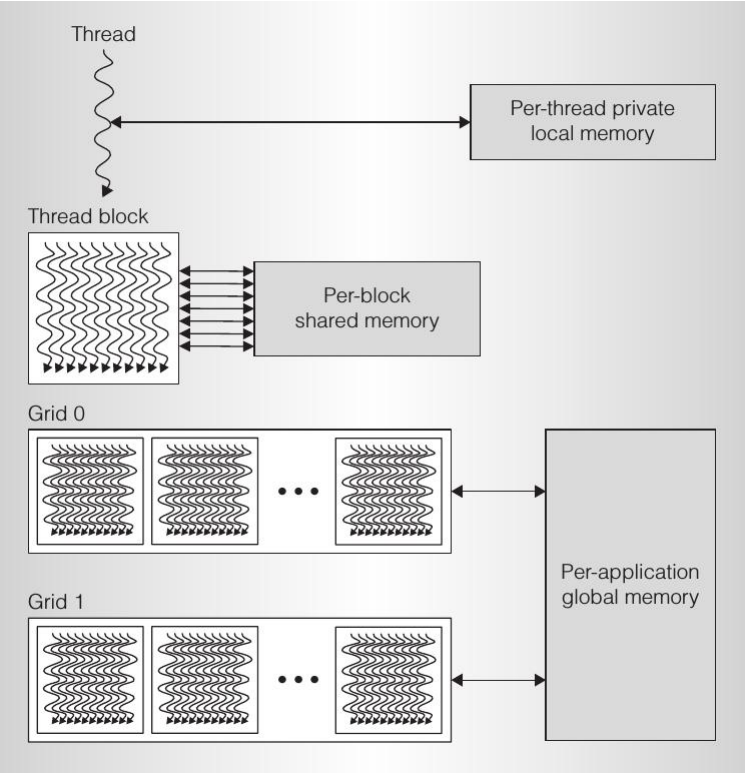

FIGURA 2.2: Schema della gerarchia di *Thread*, *Thread block*, e *Grid* [1, p. 59]

Le procedure che vengono eseguite sulla GPU vengono chiamate *Kernel* (identificabili grazie al prefisso *__global__*) ed è possibile specificarne la dimensionalità, ovvero decidere quanti *Thread*, *Thread block* e *Grid* verranno assegnati all'esecuzione del codice invocato.

```
1  __global__
2  void
3  my_kernel()
4  {
5      // GPU code...
6  }
7
8  // CPU code
9  int main()
10 {
11     int n_blocks = 8; // Example blocks number
12     int n_threads_per_block = 32; // Example threads per block number
13
14     // Invoke Kernel on GPU
15     my_kernel<<<n_blocks, n_threads_per_block>>>();
16 }
```

LISTING 2.1: Esempio di invocazione GPU kernel

Come è possibile notare nell'esempio di codice 2.1, stiamo invocando l'esecuzione di un kernel specificando l'utilizzo di 8 *Thread Block* e 32 *Thread* per blocco (se non viene specificato il numero delle *Grid* il valore di default è 1). In generale il numero totale di *Thread* che verranno utilizzati per la computazione del kernel è

$$\Gamma * B * T$$

dove Γ, B, T sono rispettivamente il numero di *Grid*, il numero di *Thread Block* e il numero di *Thread* che vogliamo utilizzare.

La possibilità di decidere la suddivisione del kernel tra griglie e blocchi ci ritorna molto utile nell'elaborazione di strutture dati non particolarmente complesse come possono essere gli array. Infatti è sufficiente invocare un kernel con numero di thread uguale (o maggiore) al numero di elementi dell'array e computare ogni elemento in un thread diverso.
Esiste però un limite al numero di *Thread* per blocco che è possibile dichiarare (nelle nuove versioni di CUDA è *1024*), dunque per ottenere l'*id* globale del thread relativo al kernel eseguito dobbiamo avvalerci anche del numero di *Grid* e *Thread Block* richiesti, secondo la seguente relazione:

$$\tau + (\beta * B) * (\gamma * G)$$

dove Γ, B, T sono rispettivamente il numero di *Grid*, il numero di *Thread Block*, e il numero di *Thread* che vogliamo utilizzare e γ, β, τ sobo gli *id* locali associati alle *Grid*, *Thread Block* e *Thread* con $\gamma < \Gamma, \beta < B, \tau < T$.

```
1  __global__
2  void
3  my_kernel(int* array)
4  {
5      // Computing current thread global id
6      int id = threadIdx.x + (blockIdx.x * blockDim.x) * (gridIdx.x * gridDim.x);
7
8      // Work with array values
9      array[id] = 0;
10 }
```

Listing 2.2: Esempio di calcolo GPU kernel thread global id

2.2 Modellazione del problema

2.3 Algoritmo parallelo

2.3.1 Acceleranti

Parallelismo dinamico

Bounding

Inferenza dei parametri

2.3.2 Gestione della memoria

2.3.3 Calcolo dell'istogramma finale

Capitolo 3

Risultati

3.1 Confronto CUDA/Python

3.2 Performance

Capitolo 4

Conclusioni

Bibliografia

[1] John Nickolls e William J Dally. “The GPU computing era”. In: *IEEE micro* 30.2 (2010).