



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

<b>Elenco delle Figure</b>	<b>ii</b>
<b>1 Introduzione</b>	<b>1</b>
<b>2 Metodi</b>	<b>2</b>
2.1 Architettura CUDA . . . . .	2
2.2 Modellazione del problema . . . . .	5
2.3 Algoritmo parallelo . . . . .	5
2.3.1 Acceleranti . . . . .	5
2.3.2 Gestione della memoria . . . . .	6
2.3.3 Calcolo dell'istogramma finale . . . . .	6
<b>3 Risultati</b>	<b>7</b>
3.1 Confronto CUDA/Python . . . . .	7
3.2 Performance . . . . .	7
<b>4 Conclusioni</b>	<b>8</b>
<b>Bibliografia</b>	<b>9</b>

# Elenco delle figure

2.1	Streaming Multiprocessor dell'architettura Fermi [1, p. 63] . . . . .	3
2.2	Schema della gerarchia di <i>Thread</i> , <i>Thread block</i> , e <i>Grid</i> [1, p. 59] . . . . .	4

# 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 fenomeni 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]

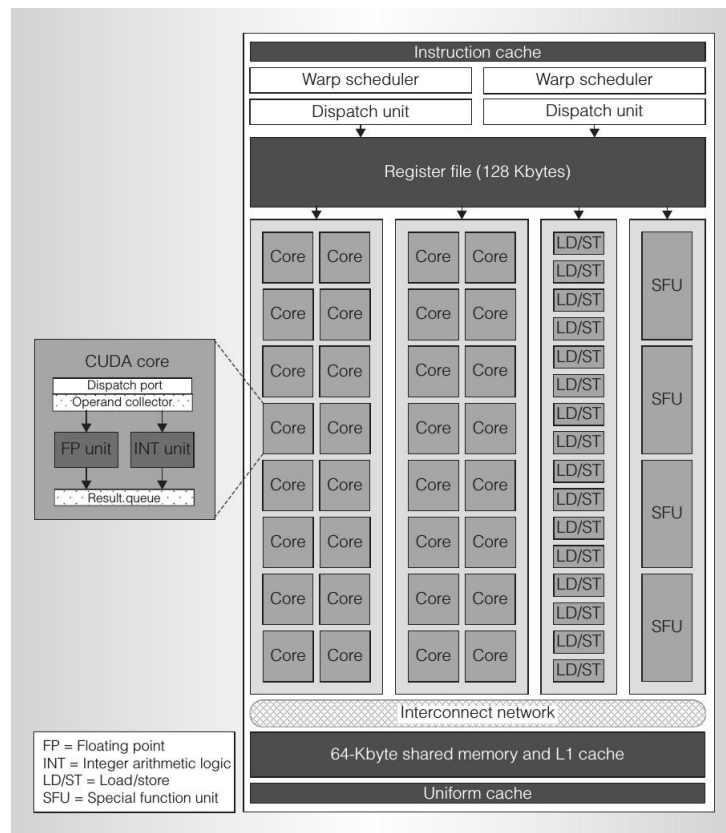


FIGURA 2.1: Streaming Multiprocessor dell'architettura Fermi [1, p. 63]

**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.

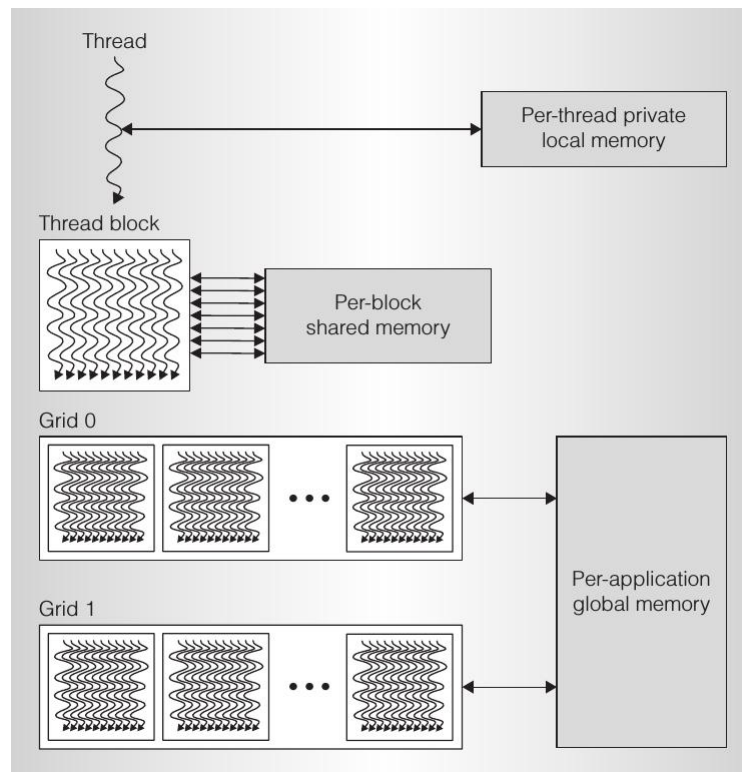


FIGURA 2.2: Schema della gerarchia di *Thread*, *Thread block*, e *Grid* [1, p. 59]

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
12     // Invoke Kernel on GPU
13     my_kernel<<<8, 32>>>();
14
15     return 0;
16 }
```

---

LISTING 2.1: 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 blocks* e 32 *Thread* per blocco (se non viene specificato il numero delle *Grid* il valore di default è 1).

Dunque in totale verranno utilizzati:

$$1 * 8 * 32 = 256$$

thread per l'esecuzione del kernel specificato.

Questa suddivisione tra thread ci ritorna molto utile nell'elaborazione di strutture dati come gli array, in quanto facilmente parallelizzabili. Infatti è sufficiente invocare un kernel con numero di thread uguale (o maggiore) al numero di elementi dell'array per ottenere il massimo grado di parallelismo

---

```
1  __global__
2  void
3  my_kernel()
4  {
5      // Computing current thread global id
6      int id = threadIdx.x + blockIdx.x * blockDim.x;
7  }
```

---

LISTING 2.2: Calcolo 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).