Corso di High Performance Computing

Esercitazione CUDA del 18/5/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/05/18

Per svolgere l'esercitazione è possibile collegarsi al server `disi-hpc-cuda.csr.unibo.it` tramite ssh, usando le credenziali fornite dal docente. Sulla macchina è installato il CUDA toolkit comprensivo di compilatore `nvcc` e alcuni editor di testo per console: `vim`, `pico`, `joe`, `ne` e `emacs`. Per chi non è pratico suggerisco `pico`, che è semplice da usare e richiede poche risorse.

Per scaricare l'archivio con i sorgenti di questa esercitazione è possibile usare i comandi:

```
wget http://www.moreno.marzolla.name/teaching/HPC/ex3-cuda.zip
unzip ex3-cuda.zip
cd ex3-cuda/
```

Ricordo che sul server è installato il comando `deviceQuery` per ottenere informazioni sulle GPU disponibili (quantità di memoria, dimensione massima della memoria condivisa, numero massimo di thread per blocco, numero di CUDA core, ecc.).
Alcuni esercizi producono immagini in formato PBM (*Portable Bitmap*) o PGM (*Portable Graymap*) che le macchine Windows dei laboratori non sono in grado di visualizzare. È necessario convertire tali immagini in un formato diverso (ad esempio, PNG) dando sul server un comando come:

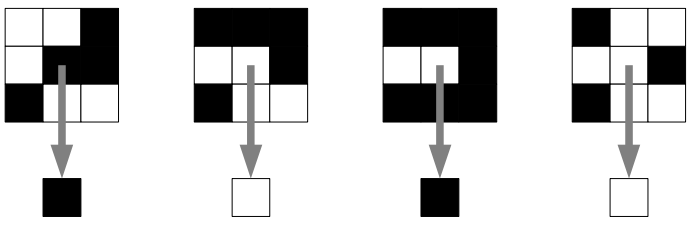
```
convert image.pbm image.png
```

per poi copiare il file risultante sul proprio PC usando il programma Winscp (già installato).

1. L'Automa cellulare "ANNEAL"

In questo esercizio consideriamo un semplice automa cellulare binario in due dimensioni, denominato ANNEAL (noto anche come *twisted majority rule*). L'automa opera su un dominio quadrato di dimensione $N \times N$, in cui ogni cella può avere valore 0 oppure 1. Si assume un dominio toroidale, in modo che ogni cella, incluse quelle sul bordo, abbia sempre otto celle adiacenti. Due celle si considerano adiacenti se hanno un lato oppure uno spigolo in comune.

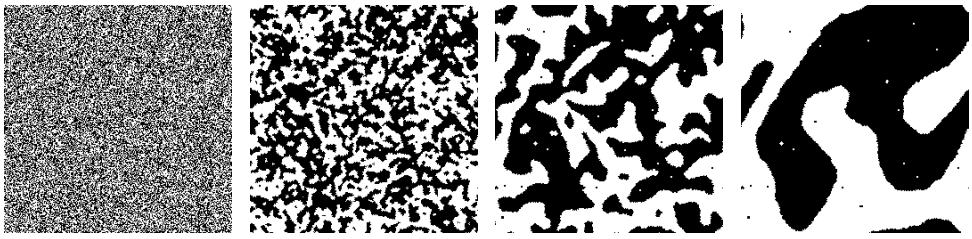
L'automa evolve a istanti di tempo discreti $t = 0, 1, 2, \ldots$. Lo stato di una cella al tempo $t + 1$ dipende dal proprio stato e da quello degli otto vicini al tempo t. in dettaglio, per ogni cella x sia B_x il numero di celle con valore 1 nell'intorno di dimensione 3×3 centrato su x (quindi si avrà sempre $0 \leq B_x \leq 9$). Se $B_x = 4$ oppure $B_x \geq 6$, allora il nuovo stato della cella x è 1; in caso contrario il nuovo stato della cella è 0. La figura seguente mostriamo alcuni esempi.



Come sempre in questi casi si devono utilizzare due griglie (domini) per rappresentare lo stato

corrente dell'automa e lo stato al passo successivo. Lo stato delle celle viene sempre letto dalla griglia corrente, e i nuovi valori vengono sempre scritti nella griglia successiva. Quando il nuovo stato di tutte le celle è stato calcolato, si scambiano le griglie e si ripete.

Il dominio viene inizializzato ponendo ogni cella a 0 o 1 con uguale probabilità; di conseguenza, circa metà delle celle saranno nello stato 0 e l'altra metà sarà nello stato 1. La figura seguente mostra l'evoluzione di una griglia di dimensione 256×256 dopo 10, 100 e 1024 iterazioni. Si può osservare come le celle 0 e 1 tendano ad addensarsi progressivamente, pur con la presenza di piccole "bolle".



Stato iniziale | Dopo 10 iterazioni | Dopo 100 iterazioni | Dopo 1024 iterazioni

Il file `cuda-anneal.cu` contiene una implementazione seriale dell'algoritmo che calcola e salva su un file l'evoluzione dopo K iterazioni dell'automa cellulare che usa la regola ANNEAL. Scopo di questo esercizio è di modificare il programma in modo da delegare alla GPU sia il calcolo del nuovo stato, sia la copia dei bordi del dominio (necessaria per simulare un dominio toroidale).

Alcuni suggerimenti:

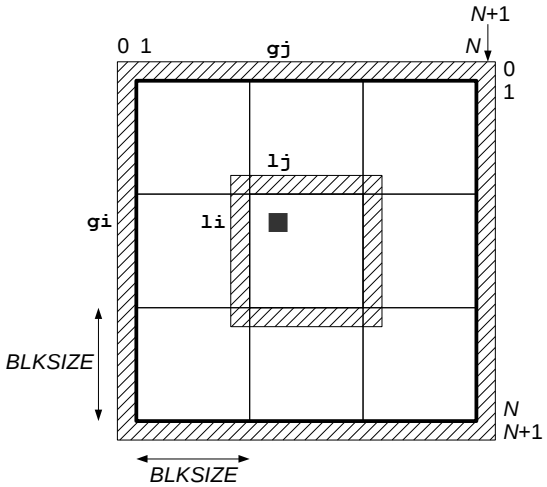
- Iniziare sviluppando una versione che non usa la memoria `__shared__`. Trasformare le funzioni `copy_top_bottom()`, `copy_left_right()` e `step()` in kernel; in questo modo è possibile fare evolvere l'automa interamente nella memoria della GPU. La dimensione dei thread block necessari a copiare le celle sarà diverso dalla dimensione dei blocchi usati per l'evoluzione dell'automa (vedi punti seguenti).
- Dato che il dominio è bidimensionale, è utile usare thread block bidimensionali per calcolare l'evoluzione delle celle. Supponendo di decomporre il dominio in thread block di dimensione $BLKSIZE \times BLKSIZE$, allora la dimensione della griglia dovrà essere $(N + BLKSIZE - 1)/BLKSIZE \times (N + BLKSIZE - 1)/BLKSIZE$ blocchi. Ricordarsi che la GPU disponibile consente al massimo 512 thread per blocco; suggerisco quindi di usare $BLKSIZE = 16$.
- Per copiare le ghost cells ai lati bastano blocchi di thread in una dimensione. Quindi per l'esecuzione dei kernel `copy_top_bottom()` e `copy_left_right()` saranno necessari $(N + 2)$ thread.
- Nel kernel `step()`, ciascun thread calcola il nuovo stato di un elemento del dominio di coordinate (i, j). Ricordare che si sta lavorando su un dominio allargato con due righe e due colonne in più, quindi le celle "vere" (non ghost) sono quelle con coordinate $1 \le i, j \le N$, Di conseguenza, ogni thread calcolerà i, j come:
  ```
  const int i = 1 + threadIdx.y + blockIdx.y * blockDim.y;
  const int j = 1 + threadIdx.x + blockIdx.x * blockDim.x;
  ```
  In questo modo i thread verranno mappati sugli elementi del dominio allargato a partire dalla posizione (1, 1) fino alla posizione (N, N). Le righe e colonne 0 e $N + 1$ rappresentano ghost cell.

Estensione (per casa, o se resta tempo)

Questo programma trarrebbe beneficio dall'uso della memoria `__shared__` (perché?). Assumiamo che *N* sia un multiplo esatto di *BLKSIZE*. Ciascun blocco di thread copia gli elementi della porzione di dominio di sua competenza in un buffer locale `buf[BLKSIZE+2][BLKSIZE+2]`, e calcolare il nuovo stato delle celle usando i dati nel buffer locale anziché accedendo alla memoria globale.

In situazioni del genere è utile usare due coppie di indici (*gi*, *gj*) per indicare le posizioni delle celle nella matrice globale e (*li*, *lj*) per indicare le posizioni delle celle nel buffer locale. L'idea è che la cella di coordinate (gi, gj) nella matrice globale corrisponda a quella di coordinate (li, lj) nel buffer locale. Usando ghost cell sia a livello globale il calcolo delle coordinate può essere effettuato come segue:

```
const int gi = 1 + threadIdx.y + blockIdx.y * blockDim.y;
const int gj = 1 + threadIdx.x + blockIdx.x * blockDim.x;
const int li = 1 + threadIdx.y;
const int lj = 1 + threadIdx.x;
```



La parte più laboriosa è la copia dei dati dalla griglia globale al buffer locale. Ricordiamo che stiamo utilizzando *BLKSIZE* × *BLKSIZE* thread per blocco. La copia della parte centrale di ciascun buffer (cioè tutto ad esclusione dell'area tratteggiata della figura sopra, che rappresenta le ghost area del dominio globale e del buffer locale) si effettua con:

```
buf[li][lj] = cur[IDX(gi, gj)];
```

Per inizializzare la ghost area si può procedere in diversi modi. Quello più semplice (e meno efficiente) è delegare l'inizializzazione al thread con (*li*, *lj*) = (1, 1), che usando opportuni cicli "for" o simili riempirà la ghost area in modo adeguato:

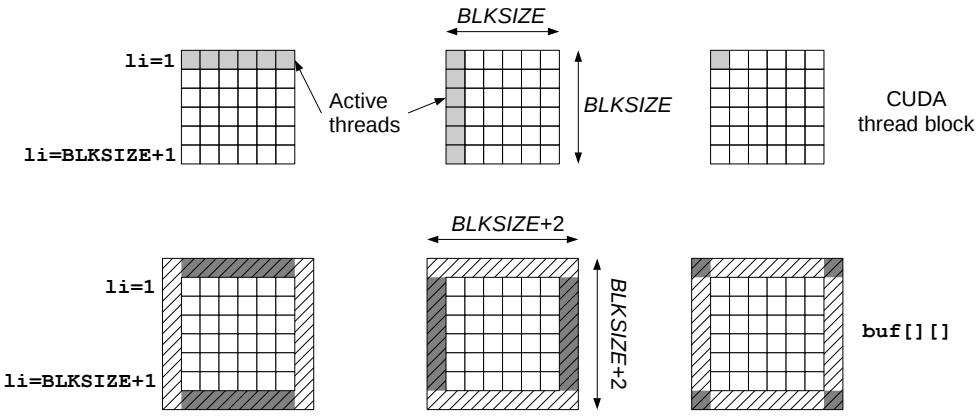
```
if ( li == 1 && lj == 1 ) {
    /* riempi tutte le celle della ghost area di buf[][] */
}
```

Una soluzione migliore è quella di delegare l'inizializzazione della ghost area ai thread della prima riga (quelli con *li* = 1) e della prima colonna (quelli con *lj* = 1); i thread della prima riga riempiranno le celle `buf[0][1..BLKSIZE]` e `buf[BLKSIZE+1][1..BLKSIZE]`, mentre quelli della prima colonna riempiranno le celle `buf[1..BLKSIZE][0]` e `buf[BLKSIZE+1][1..BLKSIZE]`: Il thread con (*li*, *lj*) = (1, 1) inizializza le quattro celle agli angoli: `buf[0][0]`,

`buf[0][BLKSIZE+1]`, `buf[BLKSIZE+1][0]`, `buf[BLKSIZE+1][BLKSIZE+1]`. In pratica, ciascun thread eseguirà il codice seguente:

```
if ( li == 1 ) {
    /* riempi la cella buf[0][lj] e buf[BLKSIZE+1][lj] */
}
if ( lj == 1 ) {
    /* riempi la cella buf[li][0] e buf[li][BLKSIZE+1] */
}
if ( li == 1 && lj == 1 ) {
    /* riempi buf[0][0] */
    /* riempi buf[0][BLKSIZE+1] */
    /* riempi buf[BLKSIZE+1][0] */
    /* riempi buf[BLKSIZE+1][BLKSIZE+1] */
}
```

La figura seguente mostra quali thread sono attivi durante le tre fasi (parte alta), e quali dati del dominio locale `buf[][]` essi gestiscono (parte bassa).



2. Il problema dello zaino

Il problema dello zaino (*knapsack problem*) è un famiglia di problemi di ottimizzazione combinatoria. In questo esercizio consideriamo la formulazione seguente, detta *problema dello zaino 0/1*: dato un insieme di n oggetti aventi pesi interi positivi $w[0], w[1], \ldots w[n-1]$ e valori reali positivi (di tipo *float*) $v[0], v[1], \ldots v[n-1]$, vogliamo determinare il massimo valore (profitto) che è possibile ottenere inserendo un opportuno sottoinsieme di oggetti in un contenitore (zaino) di capienza massima C (intero positivo). In questo esercizio ci limitiamo a calcolare il valore complessivo degli oggetti appartenenti alla soluzione ottima, piuttosto che la lista degli oggetti che fanno parte della soluzione.

Il problema può essere risolto mediante la programmazione dinamica come segue. Per ogni $i = 0, \ldots n-1$ e per ogni $j = 0, \ldots C$, sia $P[i][j]$ il massimo profitto che è possibile ottenere scegliendo un opportuno sottoinsieme dei primi $i+1$ oggetti $\{0, 1, \ldots i\}$ da inserire in uno zaino di capienza massima j. I valori $P[i][j]$ possono essere calcolati utilizzando le regole seguenti:

$$P[0][j] = \begin{cases} 0 & \text{se } j < w[0] \\ v[0] & \text{altrimenti} \end{cases}$$

e, per ogni $i = 1, \ldots n, j = 0, \ldots C$,

$$P[i][j] = \begin{cases} P[i-1][j] & \text{se } j < w[i] \\ \max\{P[i-1][j],\ P[i-1][j-w[i]] + v[i]\} & \text{altrimenti} \end{cases}$$

E' possibile calcolare i valori $P[i][j]$ procedendo per righe, iniziando dalla riga 0. Il massimo profitto ottenibile inserendo un opportuno sottoinsieme di oggetti nello zaino di capacità C è $P[n - 1][C]$.

Dato che siamo interessati a conoscere solo il valore $P[n - 1][C]$, non è necessario mantenere in memoria tutta la matrice P: come utile esercizio di teoria si consiglia di identificare le dipendenze tra i valori $P[i][j]$, verificando che ogni riga della matrice possa essere calcolata usando solo i valori della riga precedente. È quindi possibile ridurre lo spazio richiesto per il calcolo della soluzione ottima da $O(nC)$ a $O(C)$ utilizzando solo due righe della matrice, la riga corrente e la riga immediatamente successiva.

Viene fornito il file `cuda-knapsack.cu` che risolve il problema dello zaino usando solo la CPU. Il programma legge una istanza del problema da un file il cui nome deve essere passato come unico parametro sulla riga di comando, e visualizza su standard output il valore massimo degli oggetti che è possibile inserire nello zaino. Il file di input ha una struttura molto semplice: le prime due righe contengono i valori di C ed n, rispettivamente; seguono n righe ciascuna delle quali contenente il peso $w[i]$ (intero) e il valore $v[i]$ (reale) dell'oggetto i-esimo. Il programma `knapsack-gen.c` può essere usato per generare altre istanze di input.

Modificare il programma fornito definendo un kernel CUDA per il calcolo delle righe della matrice P (le parti che possono essere parallelizzate sono marcate con un `[TODO]`). Poiché il kernel deve riempire una riga per volta della matrice P, occorre usare blocchi in una dimensione di CUDA thread.