Corso di High Performance Computing

Esercitazione CUDA del 12/5/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/05/12

Per svolgere l'esercitazione è possibile collegarsi tramite ssh al server `disi-hpc-cuda.csr.unibo.it` 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/ex2-cuda.zip
unzip ex2-cuda.zip
cd ex2-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 degli 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 il comando:

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

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

1. Somma di matrici

Il file `cuda-matsum.cu` calcola la somma tra due matrici quadrate di dimensione $N \times N$ utilizzando la CPU. Modificare il file per calcolare la somma usando CUDA. Il programma deve funzionare con qualsiasi valore di N, che pertanto non deve necessariamente essere un multiplo della dimensione dei thread block.

2. La “regola 30” colpisce ancora

Questo esercizio chiede di implementare mediante CUDA l'automa cellulare della "regola 30" già visto nelle esercitazioni MPI. Per comodità riportiamo qui sotto la descrizione del funzionamento dell'automa.

Lo spazio degli stati è costituito da un array $a[N]$ di N interi, ciascuno dei quali può avere valore 0 oppure 1. L'automa evolve a passi discreti: lo stato di ogni cella al tempo t dipende esclusivamente dal proprio stato e da quello dei due immediati vicini al tempo t - 1. Assumiamo un dominio ciclico, per cui i vicini della cella $a[0]$ sono $a[N - 1]$ e $a[1]$.

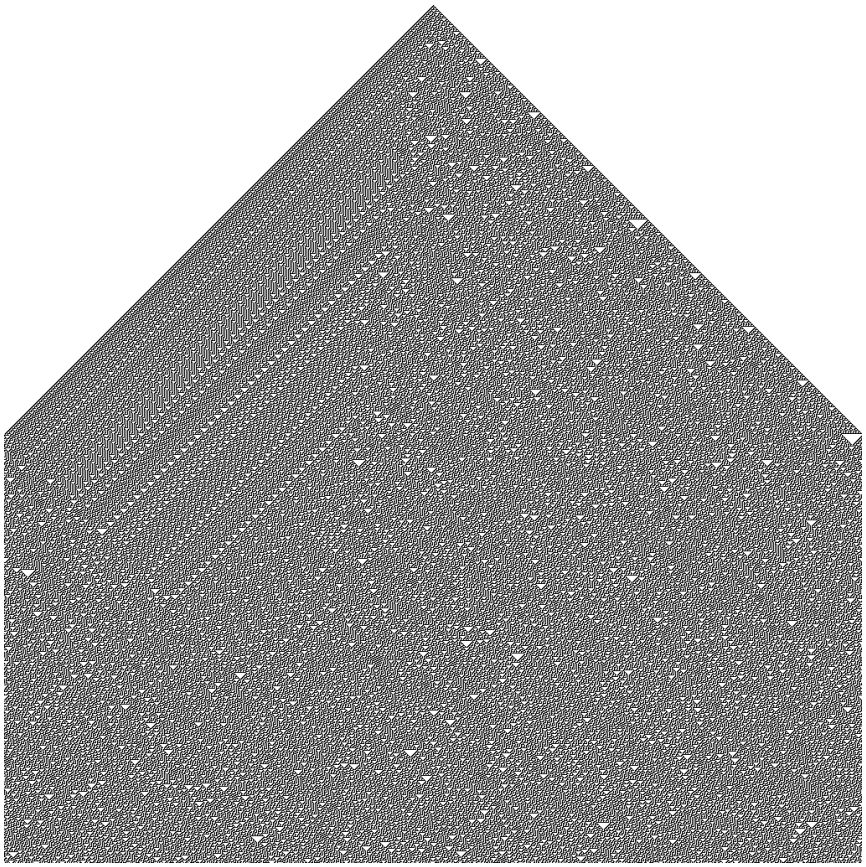
Dati i valori correnti *pqr* di tre celle adiacenti, il nuovo valore *q*' della cella centrale è determinato in base alla tabella seguente:

Configurazione corrente (*pqr*)	111	110	101	100	011	010	001	000

Nuovo stato della cella centrale (q')	0	0	0	1	1	1	1	0

(si noti che la sequenza 00011110 rappresenta il numero 30 in binario, da cui il nome "regola 30").

Il file `cuda-rule30.cu` contiene una versione seriale dell'algoritmo che calcola l'evoluzione dell'automa della regola 30. Inizialmente tutte le celle valgono 0, ad eccezione di quella in posizione $N/2$ che ha valore 1. Il programma accetta sulla riga di comando la dimensione del dominio N e il numero di passi da calcolare. Al termine dell'esecuzione il programma produce un file `rule30.pbm` il cui contenuto dovrebbe essere simile all'immagine seguente:



Scopo di questo esercizio è di svilupparne una versione parallela in cui il calcolo dei nuovi stati (cioè di ogni riga dell'immagine) venga realizzato da CUDA threads. In particolare, la funzione `rule30()` dovrà essere trasformata in un kernel che viene invocato per calcolare un singolo passo di evoluzione dell'automa. Assumere che N sia multiplo del numero di thread per blocco *BLKSIZE*.

Si consiglia di iniziare con una versione in cui si opera direttamente sulla memoria globale senza usare memoria `__shared__`; tale versione si può ricavare molto velocemente partendo dal codice seriale fornito come esempio. Da quanto detto a lezione, l'uso della memoria condivisa è molto utile in questo caso, perché lo stesso valore viene letto dalla memoria globale più volte (tre, per la precisione). Realizzare quindi una seconda versione del programma che sfrutti la memoria condivisa. La parte più complessa sarà probabilmente la copia dei valori dalla memoria globale alla memoria condivisa. Alcuni suggerimenti:

- Ciascun thread block definisce un array `__shared__` chiamato ad esempio `buf[BLKSIZE+2]` (usiamo *BLKSIZE* + 2 elementi perché dobbiamo includere una ghost cell a sinistra e a destra per poter calcolare lo stato successivo dei `BLKSIZE` elementi centrali);
- Ciascun thread determina l'indice `lindex` dell'elemento locale (nell'array `buf[]`), e lindice `gindex` dell'elemento globale (nell'array `cur[]` che denota lo stato globale dell'automa) su

cui deve operare. Supponiamo che l'array `cur[]` non includa ghost cells (è anche possibile decidere di estenderlo con due ghost cells, che andranno riempite con i valori della cella appropriata ad ogni iterazione), occorre prestare attenzione che il primo indice utile, cioè corrispondente ad una cella non-ghost, per `buf[]` è 1, mentre il primo indice utile per `cur[]` è zero. Quindi ciascun thread calcolerà gli indici come:

```
int lindex = 1 + threadIdx.x;
int gindex =  threadIdx.x + blockIdx.x * blockDim.x;
```

- Ciascun thread copia un elemento dalla memoria globale alla memoria condivisa:

  ```
  buf[lindex] = cur[gindex];
  ```

  È però necessario riempire anche le ghost cell di `buf[]`. Se si decide di non estendere `cur[]` con ghost cells, bisogna fare attenzione ai casi particolari in cui `gindex` = 0 oppure `gindex` = N - 1.

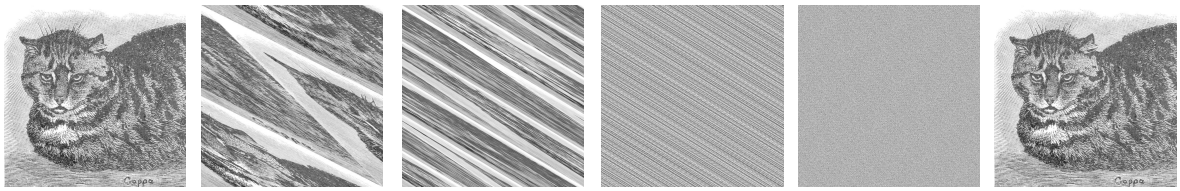
Si faccia riferimento all'esempio `cuda-stencil1d-shared.cu`, considerando però che questo esercizio è piuttosto diverso e richiede diversi adattamenti.

3. Il ritorno della mappa del gatto

Abbiamo già visto la *mappa del gatto di Arnold* nelle esercitazioni OpenMP. In questo esercizio si chiede di realizzare un programma CUDA che trasforma una immagine mediante la mappa del gatto. Per comodità riportiamo nel seguito la descrizione dettagliata del problema.
La mappa del gatto trasforma una immagine quadrata P di dimensione $W \times H$ in una nuova immagine P' delle stesse dimensioni: il pixel di coordinate (x, y) in P, $0 \le x < W$, $0 \le y < H$, viene collocato nella posizione (x', y') di P' dove:

$$x' = (2x + y) \bmod W, \quad y' = (x + y) \bmod H$$

(mod è l'operatore modulo, corrispondente all'operatore `%` del linguaggio C). Si può assumere che le coordinate (0, 0) indichino il pixel in alto a sinistra e le coordinate (W - 1, Y - 1) quello in basso a destra, in modo da poter rappresentare l'immagine come una matrice.
La mappa del gatto ha proprietà sorprendenti. Applicata ad una immagine, se ne ottiene una versione molto distorta. Applicando nuovamente la mappa a quest'ultima immagine, se ne ottiene un'altra ancora più distorta, e così via. Tuttavia, dopo un certo numero di iterazioni (il cui valore dipende dalla dimensione dell'immagine, e nel caso di immagini quadrate di dimensione $N \times N$ risulta sempre minore o uguale a $3N$) ricompare l'immagine di partenza!



Originale $K = 1$ $K = 2$ $K = 5$ $K = 100$ $K = 684$

Viene fornito un programma sequenziale che calcola la k-esima iterata della mappa del gatto usando la CPU. Il programma viene invocato specificando sulla riga di comando il numero di iterazioni k. Il programma legge una immagine in formato PGM da standard input, e produce una nuova immagine su standard output ottenuta applicando k volte la mappa del gatto. Occorre ricordarsi di redirezionare lo standard output su un file, come indicato nelle istruzioni nel sorgente. Con l'immagine data, dopo k = 684 iterazioni si deve ottenere l'immagine di partenza.

Per sfruttare il parallelismo offerto da CUDA è utile usare una griglia bidimensionale di thread block, ciascuno con $BLKSIZE \times BLKSIZE$ thread. Pertanto, data una immagine di $W \times H$ pixel, sono necessari:

$(W + BLKSIZE - 1)/BLKSIZE \times (H + BLKSIZE - 1)/BLKSIZE$

blocchi di dimensione $BLKSIZE \times BLKSIZE$.
Ogni thread si occupa di calcolare una singola iterazione della mappa del gatto, copiando un pixel dell'immagine corrente nella posizione appropriata della nuova immagine. La segnatura del kernel sarà:

```
__device__ void
  cat_map_iter( unsigned char *cur, unsigned char *next, int w, int h )
```

(dove w e h sono la larghezza e altezza dell'immagine). Utilizzando il proprio ID e quello del blocco in cui si trova, ogni thread determina le coordinate (x, y) del pixel su cui operare, e calcola le coordinate (x', y') del pixel dopo l'applicazione di una iterazione della mappa del gatto. Per calcolare la k-esima iterata sarà quindi necessario invocare il kernel k volte, scambiando dopo ogni iterazione le immagini corrente e successiva come fatto dal programma seriale.
La soluzione precedente consente di parallelizzare il programma fornito apportando minime modifiche. Esiste però anche la possibilità di definire un kernel che calcoli direttamente la k-esima iterata della mappa del gatto con una singola invocazione. La segnatura del nuovo kernel sarà

```
__device__ void
  cat_map_iter_k( unsigned char *cur, unsigned char *next, int w, int h, int k )
```

Come nel caso precedente, ciascun thread determina le coordinate (x, y) del pixel di sua competenza. Le coordinate del pixel dopo k iterazioni della mappa del gatto si possono ottenere applicando lo schema seguente:

```
const int x = ...;
const int y = ...;
int xcur = x, ycur = y, xnext, ynext;

if ( x < w && y < h ) {
     while (k--) {
          xnext = (2*xcur + ycur) % w;
          ynext = (xcur + ycur) % h;
          xcur = xnext;
          ycur = ynext;
     }
     /* copia il pixel di coordinate (x, y) dell'immagine corrente
        nelle coordinate (xnext, ynext) della nuova immagine */
}
```

In questo modo è sufficiente una singola invocazione del kernel (anziché k come nel caso precedente) per ottenere l'immagine finale. Chi lo desidera può misurare i tempi di esecuzione delle due alternative per capire se e di quanto la seconda soluzione è più efficiente della prima.