Corso di High Performance Computing

Esercitazione CUDA del 11/12/2018

Moreno Marzolla

Ultimo aggiornamento: 2018-12-05

Per svolgere l'esercitazione è necessario collegarsi al server `isi-raptor03.csr.unibo.it` tramite ssh, usando come nome utente il proprio indirizzo di posta istituzionale completo, e come password la propria password istituzionale (cioè quella usata per accedere ad AlmaEsami). Sulla macchina è installato il CUDA toolkit versione 8.0 comprensivo di compilatore `nvcc`.

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).
Chi si collega al server da una macchina Linux con ssh:

```
ssh -Y nome.cognomeNN\@studio.unibo.it@isi-raptor03.csr.unibo.it
```

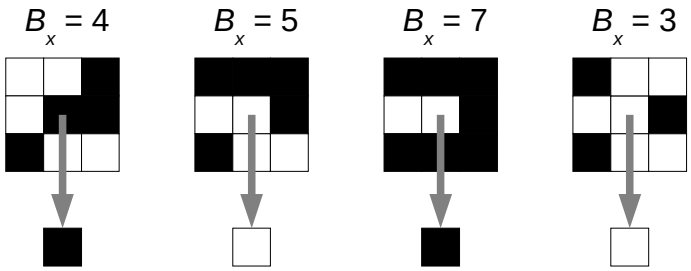
può visualizzare l'immagine senza doverla copiare localmente dando (sul server) il comando

```
display image.pbm
```

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 bidimensionale 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 vicini; 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 presenti nell'intorno di dimensione 3×3 centrato su x (si conta anche lo stato di 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 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 in modo pseudocasuale. La figura 1 mostra l'evoluzione di un dominio di dimensione 256×256 dopo 10, 100 e 1024 iterazioni. Si può osservare come le celle 0 e 1 tendano progressivamente ad addensarsi, pur con la presenza di piccole "bolle".

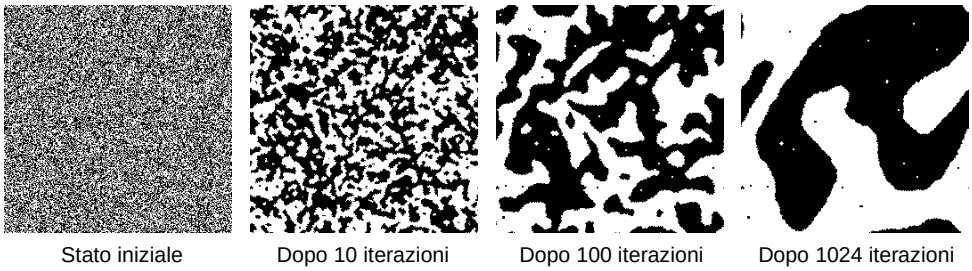


Figura 1: Evoluzione dell'automa cellulare "ANNEAL"

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

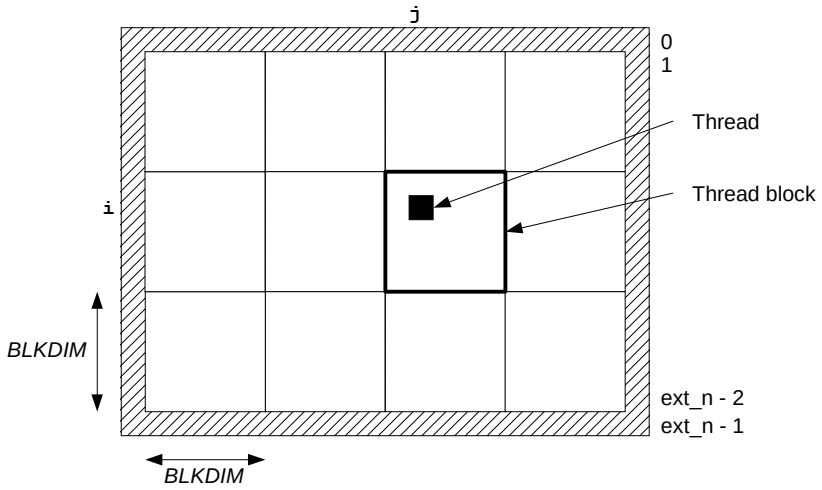
Alcuni suggerimenti:

- 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.
- 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.
- Dato che il dominio è bidimensionale, per calcolare l'evoluzione dell'automa è opportuno usare thread block bidimensionali. Supponendo di decomporre il dominio in blocchi di dimensione $BLKDIM \times BLKDIM$, allora saranno necessari $(n + BLKDIM - 1)/BLKDIM \times (n + BLKDIM - 1)/BLKDIM$ blocchi, ciascuno composto da $BLKDIM \times BLKDIM$ thread. Ricordarsi che la GPU disponibile consente al massimo 1024 thread per blocco; suggerisco quindi di porre $BLKDIM = 32$ in modo che un blocco sia composto esattamente da 1024 CUDA 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. Se definiamo $ext\_n = n + 2$ la dimensione del dominio esteso, le celle "vere" (non ghost) sono quelle con coordinate $1 \leq i, j \leq ext\_n - 2$, 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;
```

(Si faccia riferimento alla figura).



Estensione

Questo programma può trarre beneficio dall'uso della memoria `__shared__` (perché?). In questo caso, però, la gestione della memoria shared può diventare abbastanza laborioso, per cui occorre prestare attenzione. L'approccio descritto nel seguito non è l'unico possibile, ma ha il vantaggio di essere ragionevolmente semplice da implementare.

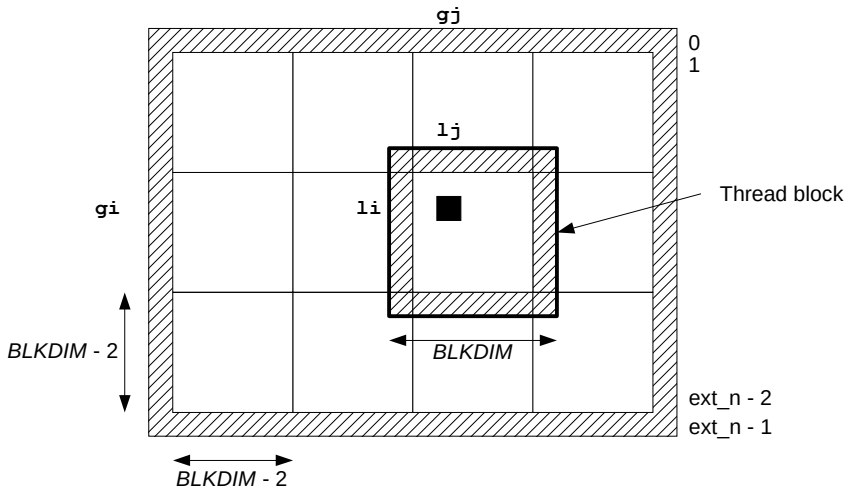


Figura 2: Automa "ANNEAL" con shared memory

Iniziamo osservando che è necessario includere una ghost area anche nella memoria shared (si veda la figura 2). Di conseguenza, un thread block di dimensione $BLKDIM \times BLKDIM$ calcolerà il nuovo stato di una porzione di dominio di dimensione $(BLKDIM - 2) \times (BLKDIM - 2)$

Assumiamo per semplicità che la dimensione del dominio *n* sia un multipla di (*BLKDIM* - 2). Ciascun blocco di thread copia gli elementi della propria porzione di dominio in un buffer locale `buf[BLKDIM][BLKDIM]` che include due righe e due colonne (le prime e le ultime) di ghost cells, 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*) nel dominio globale corrisponda a quella di coordinate (*li*, *lj*) nel buffer locale. Usando ghost cell sia a livello globale che a livello locale il calcolo delle coordinate deve essere effettuato come segue:

```
const int gi = threadIdx.y + blockIdx.y * (blockDim.y - 2);
const int gj = threadIdx.x + blockIdx.x * (blockDim.x - 2);
const int li = threadIdx.y;
const int lj = threadIdx.x;
```

Si noti che le espressioni usate per *gi* e *gj* sono diverse da quelle che abbiamo visto in altre occasioni; la ragione è che la dimensione dei thread block include le ghost cells locali.

Una volta calcolati gli indici è possibile copiare i dati dalla memoria globale al buffer locale mediante l'istruzione

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

che viene eseguita da tutti i thread di ciascun blocco. Fatto questo, i thread che *non* stanno sul bordo del proprio blocco possono calcolare il nuovo stato delle celle; un thread non si trova sul bordo se si ha $0 < li < blockDim.y - 1$ e $0 < lj < blockDim.x - 1$. Il calcolo del nuovo stato di una cella si effettua allo stesso modo della versione seriale fornita.

Come ulteriore estensione, si modifichi il codice per farlo funzionare anche nel caso in cui la dimensione *n* del dominio non sia un multiplo esatto di (*BLKDIM* - 2).

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 $i = 1, \ldots n, j = 0, \ldots C$,

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

È 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.