Corso di High Performance Computing

Esercitazione CUDA del 24/11/2020

Moreno Marzolla

Ultimo aggiornamento: 2020-11-10

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 comprensivo di compilatore `nvcc`.

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

```
wget https://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 (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 la funzione `matsum()` in modo che la somma venga realizzata dalla GPU, senza necessità di modificare il corpo principale del programma. In altre parole, la funzione `matsum()` dovrebbe:

- allocare la memoria nella GPU per memorizzare copie delle matrici *p*, *q*, *r*;
- copiare il contenuto delle matrici *p*, *q* dall'*host* al *device*;
- eseguire un apposito kernel (da definire) per calcolare la somma *p* + *q* usando la GPU;
- copiare il risultato dal *device* all'*host*;
- liberare la memoria nel *device*

in modo che non sia necessario modificare la funzione `main()`.
Il programma deve funzionare con qualsiasi valore di *N*, che quindi non deve necessariamente essere un multiplo della dimensione dei thread block CUDA. Si noti che in questo esercizio non è conveniente usare la shared memory (perché?).

2. Il ritorno dell'automa cellulare della "regola 30"

Questo esercizio chiede di implementare mediante CUDA l'automa cellulare della "regola 30" che abbiamo già incontrato in una precedente esercitazione. Per comodità riportiamo 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, in modo che tutte le celle (incluse quelle ai bordi) abbiano sempre entrambi i vicini.

Dati i valori correnti *pqr* di tre celle adiacenti, il nuovo valore *q*' della cella centrale è determinato in base alla tabella seguente (■ = 1, □ = 0):

Configurazione corrente (*pqr*)	■■■	■■□	■□■	■□□	□■■	□■□	□□■	□□□
Nuovo stato della cella centrale (*q*')	□	□	□	■	■	■	■	□

(si noti che la sequenza □□□■■■■□ = 00011110, che si legge sulla seconda riga della tabella precedente, rappresenta il valore 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 contengono 0, ad eccezione di quella in posizione *N*/2 che contiene 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 alla figura 1.

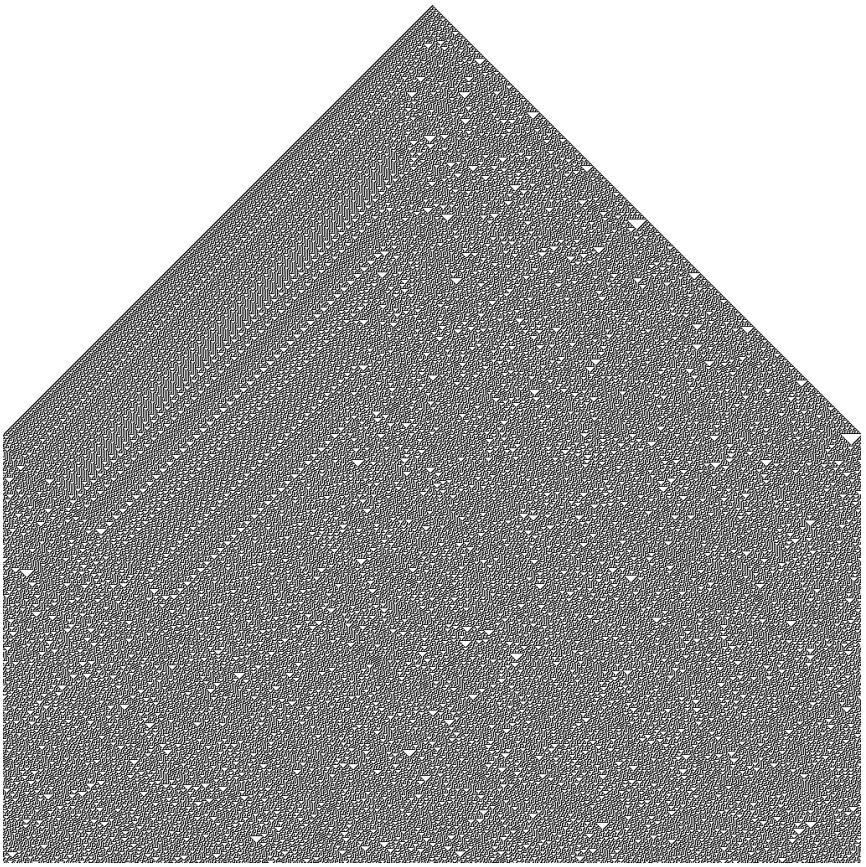


Figura 1: Evoluzione dell'automa della "regola 30" partendo da una singola cella attiva al centro del dominio

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 passo di evoluzione dell'automa. Assumere che *N* sia multiplo del numero di thread per blocco (*BLKDIM*).

Si consiglia di iniziare con una versione in cui si opera direttamente sulla memoria globale senza usare memoria condivisa (`__shared__`); questa prima versione si può ricavare molto velocemente partendo dal codice seriale fornito come esempio.

Da quanto detto a lezione, l'uso della memoria condivisa può essere utile perché ogni valore di stato nella memoria globale viene letto più volte (tre, per la precisione). Realizzare una seconda versione del programma che sfrutti la memoria condivisa (*shared*). L'idea è la stessa dell'esempio visto a lezione relativo ad una computazione di tipo *stencil*, con la differenza che in questo caso il raggio della *stencil* vale 1, cioè il nuovo stato di ogni cella dipende dallo stato precedente di quella cella e dei due vicini. Si presti però attenzione che, a differenza dell'esempio visto a lezione, qui si assume un dominio ciclico. La parte più delicata sarà quindi la copia dei valori dalla memoria globale alla memoria condivisa.

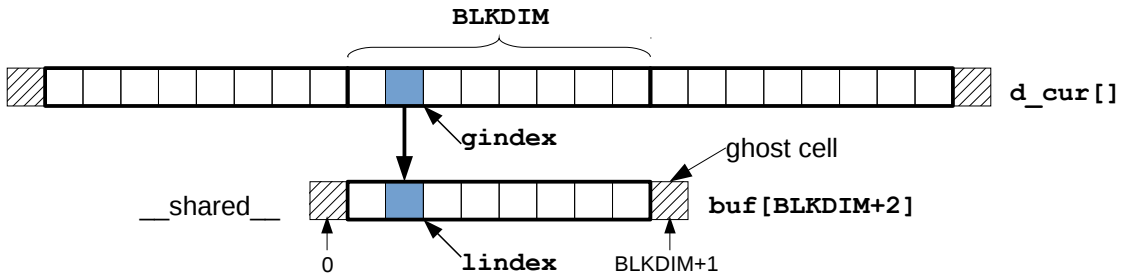


Figura 2: Copia dalla memoria globale alla memoria condivisa

Aiutandoci con la figura 2 procediamo come segue:

- `d_cur[]` è la copia nella memoria del device dello stato corrente dell'automa.
- Definire un kernel per riempire le ghost cells di `d_cur[]`; tale kernel, chiamato ad esempio `fill_ghost(...)`, deve essere eseguito da un singolo thread e quindi andrà invocato come `fill_ghost<<<1, 1>>>( … )`
- Definire un secondo kernel 1D per aggiornare il dominio. Ciascun thread block definisce un array `__shared__` chiamato ad esempio `buf[BLKDIM+2]`; usiamo *BLKDIM* + 2 elementi perché dobbiamo includere una ghost cell a sinistra e una a destra per calcolare lo stato successivo dei *BLKDIM* elementi centrali senza dover ricorrere a ulteriori letture dalla memoria globale;
- Ciascun thread determina l'indice `lindex` dell'elemento locale (nell'array `buf[]`), e l'indice `gindex` dell'elemento globale (nell'array `cur[]` in memoria globale) su cui deve operare. Estendiamo l'array `cur[]` con due ghost cells (una per estremità), come pure l'array `buf[]` in memoria condivisa. Quindi gli indici vanno calcolati come:

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

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

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

- Il primo thread di ciascun blocco inizializza le ghost cells di `buf[]` ad esempio con:

  ```
  if (0 == threadIdx.x) {
      buf[0] = cur[gindex-1];
      buf[BLKDIM + 1] = cur[gindex + BLKDIM];
  }
  ```

Poiché vogliamo generare la figura che mostra l'evoluzione dell'automa, le nuove configurazioni vanno trasferite dal *device* all'*host* al termine di ogni esecuzione del kernel.

3. Il ritorno della mappa del gatto

Anche la *mappa del gatto di Arnold* è una vecchia conoscenza che abbiamo incontrato in una delle esercitazioni OpenMP. In questo esercizio si chiede di realizzare un programma CUDA che trasforma una immagine mediante la mappa del gatto. Riportiamo nel seguito la descrizione del problema.
La mappa del gatto trasforma una immagine quadrata P di dimensione $N \times N$ in una nuova immagine P' delle stesse dimensioni: il pixel di coordinate (x, y) in P, $0 \le x < N$, $0 \le y < N$, viene collocato nella posizione (x', y') di P' dove:

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

(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 (N - 1, N - 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 (figura 3). 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.

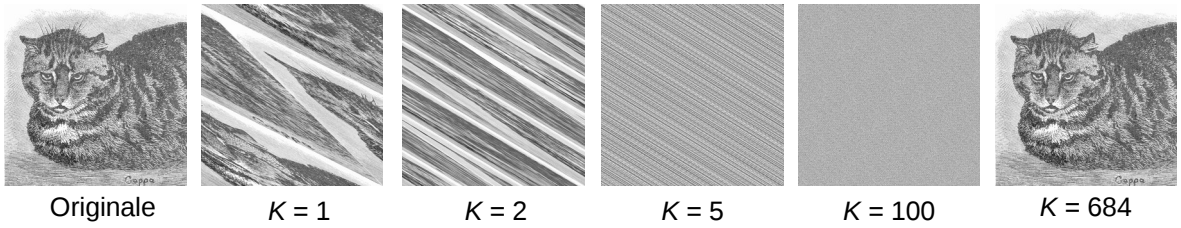


Figura 3: Esempio di applicazione iterata della mappa del gatto

Nel caso dell'immagine `cat.pgm` fornita come esempio, il tempo minimo di ricorrenza (cioè il numero minimo di iterazioni dopo le quali ricompare l'immagine originale) è 36; quindi, iterando k volte della mappa del gatto si otterrà l'immagine originale se e solo se k è multiplo di 36 (684 è multiplo di 36). Al momento non è nota alcuna relazione che lega il tempo minimo di ricorrenza alla dimensione N dell'immagine.
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.
Per sfruttare il parallelismo offerto da CUDA è utile usare una griglia bidimensionale di thread block a loro volta bidimensionali, ciascuno con $BLKDIM \times BLKDIM$ thread. Pertanto, data una immagine di $N \times N$ pixel, sono necessari:

$$(N + BLKDIM - 1) / BLKDIM \times (N + BLKDIM - 1) / BLKDIM$$

blocchi di dimensione $BLKDIM \times BLKDIM$ per ricoprire interamente l'immagine.

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:

```
__global__ void
  cat_map_iter( unsigned char *cur, unsigned char *next, int N )
```

(dove N è la larghezza o altezza dell'immagine, che deve essere quadrata). 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. Si può anche definire un kernel che calcoli direttamente la k-esima iterata della mappa del gatto con una singola invocazione. La segnatura del nuovo kernel sarà

```
__global__ void
  cat_map_iter_k( unsigned char *cur, unsigned char *next, int N, 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 si possono ottenere applicando lo schema seguente:

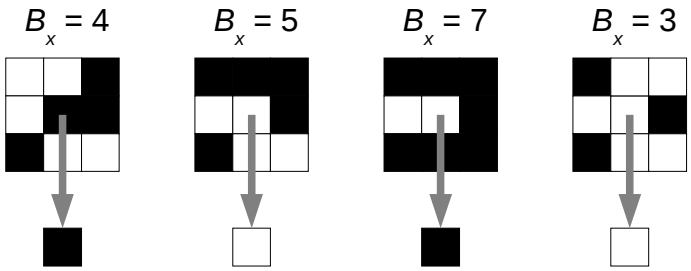
```
const int x = ...;
const int y = ...;
int xcur = x, ycur = y, xnext, ynext;

if ( x < N && y < N ) {
      while (k--) {
            xnext = (2*xcur + ycur) % N;
            ynext = (xcur + ycur) % N;
            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. Consiglio di misurare i tempi di esecuzione delle due alternative per capire se e di quanto la seconda soluzione è più efficiente della prima.

4. 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 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 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 4 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 progressivamente ad addensarsi, pur con la presenza di piccole "bolle".

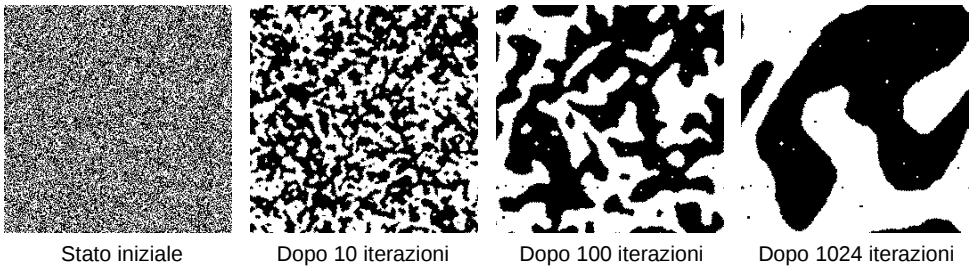


Figura 4: Evoluzione dell'automa "ANNEAL"

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 basato sulla regola ANNEAL. Scopo di questo esercizio è di modificare il programma per 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 opportuno usare thread block bidimensionali per calcolare l'evoluzione delle celle. Supponendo di decomporre il dominio in blocchi 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 1024 thread per blocco; suggerisco quindi di usare *BLKSIZE* = 32 in modo che un blocco sia composto esattamente da 1024 CUDA thread.
- Per copiare le ghost cells ai lati bastano blocchi di thread 1D. 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 $1 \leq i, j \leq N$, 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.

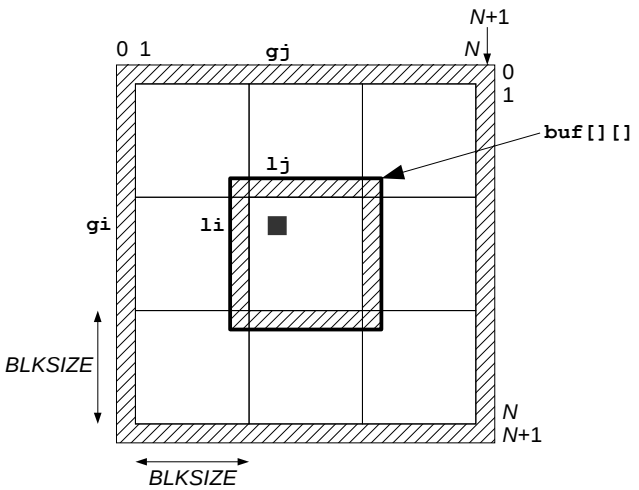
Estensione (richiede tempo perché è laboriosa)

Questo programma potrebbe trarre beneficio dall'uso della memoria `__shared__`; in realtà non c'è alcun beneficio sulle GPU installate sul server, perché sono dotate di memoria cache, e il numero di "riletture" delle celle di memoria non è sufficientemente alto da ammortizzare il costo della copia dei dati dalla memoria globale alla memoria shared. Nonostante questo, è comunque istruttivo tentare di realizzare una versione che sfrutti la memoria shared per rendersi conto di quanto possa essere laboriosa.

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]` 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) nella matrice globale corrisponda a quella di coordinate (li, i) nel buffer locale. Usando ghost cell sia a livello globale che a livello locale 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;
```



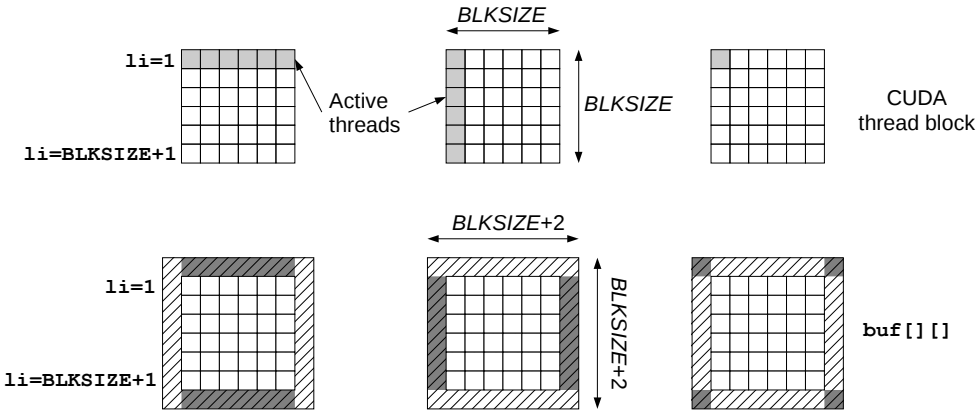
La parte più laboriosa di questa versione del programma è la copia dei dati dalla griglia globale al buffer locale. Supponiamo di utilizzare $BLKSIZE \times BLKSIZE$ thread per blocco. La copia della parte centrale di ciascun buffer (cioè tutto ad esclusione dell'area tratteggiata della figura sopra, che rappresenta le ghost area del dominio globale e del buffer locale) si effettua con:

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

Per inizializzare la ghost area si può procedere in diversi modi. Una possibile soluzione 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 la ghost area in alto e in basso, cioè le celle `buf[0][1..BLKSIZE]` e `buf[BLKSIZE+1][1..BLKSIZE]`. I thread della prima colonna riempiranno la ghost area a sinistra e a destra, cioè 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).



Sono ovviamente possibili anche altre soluzioni per riempire le ghost cells del buffer locale, ad esempio delegando la copia degli elementi solo alla prima riga (o alla prima colonna) di thread.

Infine, chi desidera cimentarsi con una versione ancora più complessa può modificare il codice per farlo funzionare anche nel caso in cui la dimensione *N* del dominio non sia un multiplo esatto di *BLKSIZE*. Questo è meno semplice di quanto sembri: non è sufficiente “disattivare” i thread al di fuori del dominio, ma bisogna prestare attenzione a come si riempiono le ghost cells dei thread block sui bordi del dominio…

5. N-Body Simulation

Nella prima lezione del corso abbiamo visto un video della simulazione *Bolshoi* (http://hipacc.ucsc.edu/Bolshoi.html), il cui scopo è di studiare l'evoluzione su larga scala dell'universo. Simulazioni di questo tipo si basano, tra le altre cose, sulla soluzione del “problema degli *N* corpi”, che consiste nello studio della dinamica di *N* masse nello spazio soggette alla mutua

attrazione gravitazionale. La simulazione Bolshoi ha richiesto 6 milioni di ore di CPU su uno dei supercomputer più potenti disponibili all'epoca. In questo esercizio ci accontentiamo di risolvere il problema degli *N* corpi usando un algoritmo molto semplice, basato su un programma sviluppato da Mark Harris liberamente disponibile all'indirizzo https://github.com/harrism/mini-nbody (il programma proposto in questo esercizio è una versione modificata dell'originale).

Le leggi fisiche che governano la dinamica di *N* masse puntiformi nello spazio sono state scoperte da Isaac Newton: si tratta della seconda legge della dinamica, e della legge di gravitazione universale. La seconda legge della dinamica afferma che la forza *F* che agisce su una particella di massa *m* le imprime una accelerazione *a* tale che $F = ma$. La legge di gravitazione universale afferma che due masse m_1 e m_2 che si trovano a distanza *d* subiscono una forza attrattiva di magnitudine $F = Gm_1m_2/d^2$ essendo G la costante di gravitazione universale ($G = 6{,}674 \times 10^{-11}$ N m^2 kg^{-2}). La spiegazione che segue non è essenziale per svolgere il progetto (viene già fornita una implementazione seriale funzionante), ma è probabilmente istruttiva e sicuramente alla vostra portata, essendo basata su concetti di fisica di base che avete visto nel corso omonimo.

Consideriamo *N* punti di massa rispettivamente m_0, … m_{n-1} disposti nello spazio. Le masse si attraggono per effetto della forza di gravità; poiché sono puntiformi, non urtano mai l'una con l'altra. Indichiamo con (x_i, y_i, z_i) la posizione e con (vx_i, vy_i, vz_i) il vettore velocità della massa *i*-esima all'istante *t*. Per calcolare le posizioni all'istante $t + \Delta t$ procediamo secondo i passi seguenti.

1. Si calcola la forza F_i che agisce sulla massa *i*-esima nell'istante corrente come segue:

   $$F_i := \sum_{i \neq j} \frac{G m_i m_j}{d_{ij}^2} n_{ij}$$

   ove *G* è la costante di gravitazione universale ($G = 6{,}674 \times 10^{-11}$ N m^2 kg^{-2}), d_{ij} è la distanza tra le particelle *i* e *j* e n_{ij} è il vettore unitario normale che punta dalla particella *i* verso la particella *j*

2. Si calcola il vettore dell'accelerazione a_i che agisce sulla massa *i*-esima all'istante corrente come segue:

   $$a_i := F_i / m_i$$

3. Si calcola il vettore della velocità v'_i della massa *i*-esima al passo successivo come:

   $$v'_i := v_i + a_i \Delta t$$

4. Si calcola la nuova posizione x'_i della massa *i*-esima al tempo $t + \Delta t$ come:

   $$x'_i := x_i + v'_i \Delta t$$

I passi precedenti realizzano la risoluzione numerica delle equazioni del moto delle *N* masse utilizzando uno schema di Eulero. L'integrazione di Eulero non è numericamente stabile per questo problema, e per tale ragione si preferiscono schemi più complessi ma più accurati.

Nel programma che consideriamo, sacrifichiamo l'accuratezza scientifica in favore della semplicità ignorando il fattore Gm_im_j e riscrivendo la sommatoria come:

$$F_i := \sum_j \frac{d_{ij}}{(\|d_{ij}\| + \epsilon)^{3/2}}$$

dove $\|d_{ij}\|$ indica la la somma dei quadrati delle distanze lungo gli assi *x*, *y*, *z* delle particelle *i* e *j*, e il termine “epsilon” al denominatore serve a evitare una divisione per zero durante il calcolo dell'interazione tra una particella e se stessa, in cambio di un trascurabile errore in tutti gli altri casi.

Si modifichi il programma fornito per usare la GPU. Una prima versione si può ottenere facilmente partendo dalla versione fornita. Si può tuttavia sfruttare la *shared memory* per ottimizzare gli accessi alla memoria della GPU durante il calcolo delle interazioni. Infatti, i dati di ciascuna particella vengono riletti *N* volte, in quanto servono per calcolare la forza che agisce su tutte le altre *N* particelle. Può quindi essere utile ricopiare blocchi di coordinate delle particelle in un array in memoria shared da usare per il calcolo delle iterazioni (vedi figura seguente). Questo fa sì che i dati di ogni particella vengano letti una sola volta per essere copiati in memoria shared, e da qui vengano letti *N* volte, una da ciascun thread.