Corso di High Performance Computing

Esercitazione CUDA del 17/11/2020

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/ex1-cuda.zip
unzip ex1-cuda.zip
cd ex1-cuda/
```

0. Familiarizzare con l'ambiente di lavoro

Il server `isi-raptor03.csr.unibo.it` dispone di tre GPU identiche (NVidia GeForce GTX 1070). Di default viene utilizzata la prima; è però possibile selezionare la scheda da usare sia da programma (es., `cudaSetDevice(1)` usa la seconda; le GPU sono numerate a partire da zero), sia mediante la variabile d'ambiente `CUDA_VISIBLE_DEVICES`; ad esempio

```
CUDA_VISIBLE_DEVICES=0 ./cuda-stencil1d
```

esegue il programma `cuda-stencil1d` sulla prima GPU (default), mentre

```
CUDA_VISIBLE_DEVICES=1 ./cuda-stencil1d
```

lo esegue sulla seconda.

Eseguire il comando `deviceQuery` per visualizzare le caratteristiche delle GPU.

1. Prodotto scalare

Modificare il file `cuda-dot.c` per calcolare e stampare il prodotto scalare tra due array `x[]` e `y[]` di uguale lunghezza *n* sfruttando la GPU, trasformando la funzione `dot()` in un kernel. Ricordo che il prodotto scalare *s* di due array `x[]` e `y[]` è definito come

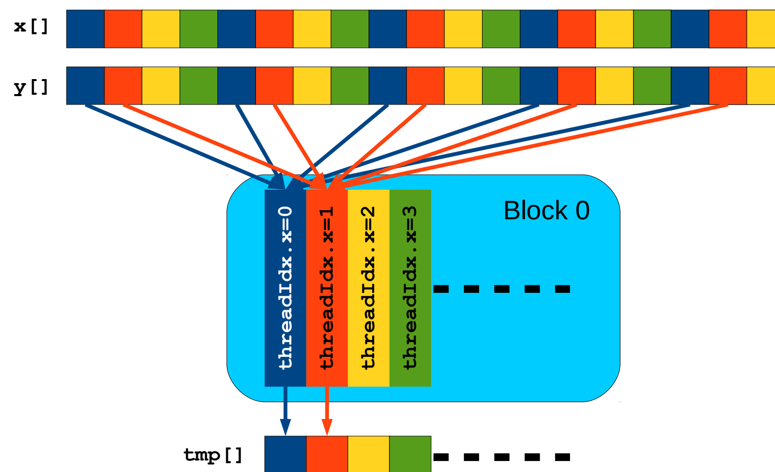
$$s=\sum_{i=0}^{n-1} x[i]\times y[i]$$

Sono necessarie diverse modifiche alla funzione `dot()` per sfruttare la GPU. Per questo esercizio si richiede di utilizzare *un singolo blocco* composto da *BLKDIM* threads, procedendo come segue:

1. La CPU alloca sulla GPU un array `tmp[]` di *BLKDIM* elementi, oltre ad una copia degli array `x[]` e `y[]`. Per allocare l'array `tmp[]` è possibile utilizzare la funzione `cudaMalloc()`.
2. La CPU attiva un singolo *thread block* 1D composto da *BLKDIM* thread
3. Il thread t (t = 0, … $BLKDIM$ - 1) calcola il valore dell'espressione ($x[t] \times y[t] + x[t + BLKDIM] \times y[t + BLKDIM] + x[t + 2 \times BLKDIM] \times y[t+2 \times BLKDIM] + \ldots$) e memorizza il risultato nell'elemento `tmp[t]`.

4. Una volta che il kernel termina l'esecuzione, la CPU trasferisce l'array `tmp[]` dalla memoria del device a quella dell'host, e ne somma il contenuto determinando così il prodotto scalare cercato.

Si rende quindi necessario calcolare il prodotto scalare in due fasi: la prima (passo 3) viene svolta dalla GPU, mentre la seconda (passo 4) viene svolta dalla CPU. La figura seguente mostra l'assegnazione del calcolo dei prodotti scalari ai thread CUDA, nel caso *BLKDIM* = 4 e *n* = 19.



Il programma deve funzionare correttamente per qualunque valore di *n*, anche se non è un multiplo di *BLKDIM*.

2. Inversione di un array

Realizzare un programma per invertire un array di *n* elementi di tipo intero, scambiando il primo elemento con l'ultimo, il secondo col penultimo e così via. È richiesta la realizzazione di due kernel distinti: il primo deve ricopiare gli elementi di un array `in[]` in un altro array `out[]` in modo che quest'ultimo contenga gli stessi elementi di in `in[]` ma in ordine inverso; il secondo kernel deve invertire gli elementi di `in[]` "*in place*", ossia modificando `in[]` senza sfruttare altri array di appoggio.

Il file `cuda-reverse.cu` fornisce una implementazione basata su CPU delle funzioni `reverse()` e `inplace_reverse()`; modificare il programma per trasformare le funzioni in kernel da invocare opportunamente.

Suggerimento: la funzione `reverse()` può essere facilmente trasformata in un kernel che viene eseguito da *n* CUDA thread (uno per ogni elemento dell'array di input). Ciascun thread si occupa di copiare un elemento di `in[]` nella corretta posizione nell'array `out[]`; utilizzare *thread block* in una dimensione, dato che in tal caso risulta facile mappare ciascun thread in un elemento dell'array di input. La funzione `inplace_reverse()` si trasforma in un kernel in modo simile, ma a differenza della precedente verrà invocata da *n*/2 CUDA thread complessivi anziché da *n*; ciascuno degli *n*/2 thread si occuperà di scambiare un elemento della prima metà dell'array `in[]` con l'elemento in posizione simmetrica nella seconda metà. Controllare che il programma funzioni anche se *n* è dispari.

3. Odd-Even Transposition Sort

A lezione è stato discusso l'algoritmo di ordinamento *Odd-Even Transposition Sort*. L'algoritmo è una variante di BubbleSort, ed è in grado di ordinare un array di *n* elementi in tempo $O(n^2)$. Pur non

essendo efficiente, l'algoritmo si presta bene ad essere parallelizzato: abbiamo già visto una versione parallela che usa OpenMP, e una versione a memoria distribuita che usa MPI. In questo esercizio viene richiesta la realizzazione di una versione CUDA.

Dato un array `v[]` di n elementi, l'algoritmo esegue n fasi numerate da 0 a $n - 1$; nelle fasi pari si confrontano gli elementi di `v[]` di indice pari con i successivi, scambiandoli se non sono nell'ordine corretto. Nelle fasi dispari si esegue la stessa operazione confrontando gli elementi di `v[]` di indice dispari con i successivi.

Il file `cuda-odd-even.cu` contiene una implementazione dell'algoritmo Odd-Even Transposition Sort. L'implementazione fornita fa solo uso della CPU: scopo di questo esercizio è di sfruttare il parallelismo CUDA mediante la definizione e l'uso di un kernel.

Il paradigma CUDA suggerisce di adottare un parallelismo a grana fine, facendo gestire ad ogni CUDA thread il confronto e lo scambio di una coppia di elementi adiacenti. La soluzione più semplice consiste nel creare n CUDA threads e lanciare n volte un kernel che sulla base dell'indice della fase (da passare come parametro al kernel), attiva i thread che agiscono sugli elementi dell'array di indice pari o quelli di indice dispari. In altre parole, la struttura di questo kernel è qualcosa del genere:

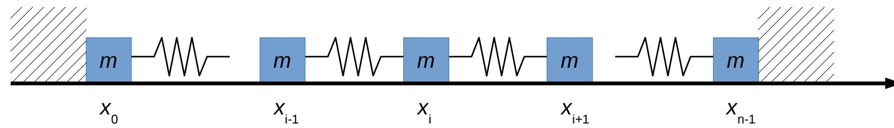
```
__global__ odd_even_step_bad( int *x, int n, int phase )
{
      const int idx = threadIdx.x + blockIdx.x * blockDim.x;
      if ( (idx < n-1) && ((idx % 2) == (phase % 2)) ) {
            cmp_and_swap(&x[idx], &x[idx+1]);
      }
}
```

Tale soluzione non è però molto efficiente, perché durante ogni fase usa solo metà dei thread disponibili. Realizzare quindi una seconda versione in cui in ogni fase si lanciano $n/2$ CUDA thread, essendo n il numero di elementi dell'array da ordinare, facendo in modo che tutti i thread siano sempre attivi durante ogni fase e gestiscano ciascuno la coppia corretta di elementi. Quindi nelle fasi pari i thread 0, 1, 2, 3, … gestiranno rispettivamente le coppie di elementi (0, 1), (2, 3), (4, 5), (6, 7), …, mentre nelle fasi dispari gestiranno rispettivamente le coppie di elementi (1, 2), (3, 4), (5, 6), (7, 8), … . Si ha quindi un kernel con la struttura seguente:

```
__global__ odd_even_step_good( int *x, int n, int phase )
{
    const int tid = threadIdx.x + blockIdx.x * blockDim.x;
    const int idx = tid*2 + (phase % 2);
    if ( idx < n-1 ) {
        cmp_and_swap( &x[idx], &x[idx+1] );
    }
}
```

4. Oscillatori accoppiati

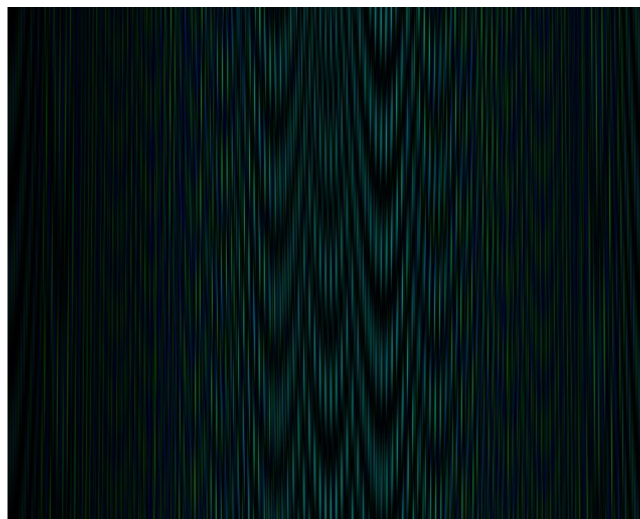
Consideriamo n punti di massa m disposti lungo una retta alle coordinate x_0, x_1, … x_{n-1}. Masse adiacenti sono collegate da una molla di costante elastica k e lunghezza a riposo L. Il primo e l'ultimo punto (quelli in posizione x_0 e x_{n-1}) occupano una posizione fissa e non possono muoversi.



Se all'istante iniziale una delle molle non è a riposo, si innescheranno delle oscillazioni che proseguiranno indefinitamente se le masse scivolano senza attrito. Sfruttando la seconda legge della dinamica di Newton $F = ma$ e la legge di Hooke che afferma che una molla di costante k compressa di una quantità Δx esercita una forza pari a $k\Delta x$, sviluppiamo un programma che, date le posizioni e le velocità iniziali delle masse, calcoli posizioni e velocità al tempo $t > 0$. Il programma si basa su un algoritmo iterativo che partendo dalle posizioni e velocità delle masse al tempo t, determina le nuove posizioni e le nuove velocità al tempo $t + \Delta t$. In particolare, la funzione `step(double *x, double *v, double *xnext, double *vnext, int n)` calcola posizione `xnext[i]` e velocità `vnext[i]` della massa i-esima al tempo $t + \Delta t$, $0 \leq i < n$, date le posizioni `x[i]` e velocità `v[i]` al tempo t.

1. Per ogni $i = 1, \ldots n - 2$ si calcola la forza F_i che agisce sulla massa i-esima come $F_i := k(x_{i-1} - 2x_i + x_{i+1})$. Le masse 0 e n - 1 rimangono in posizione fissa, quindi le forze che agiscono su di esse sono ininfluenti e non vengono calcolate.
2. Per ogni $i = 1, \ldots n - 2$ si calcola la nuova velocità v'_i della massa i-esima al tempo $t + \Delta t$ come $v'_i := v_i + (F_i / m)\Delta t$. Le masse 0 e n - 1 restano fisse, quindi la loro velocità sarà sempre zero.
3. Per ogni $i = 1, \ldots n - 2$ si calcola la nuova posizione x'_i della massa i-esima al tempo $t + \Delta t$ come $x'_i := x_i + v'_i \Delta t$. Le masse 0 e n - 1 restano ferme quindi la loro posizione al tempo $t + \Delta t$ sarà uguale a quella al tempo t: $x'_0 = x_0$. $x'_{n-1} = x_{n-1}$.

Il file `cuda-coupled-oscillators.c` contiene una versione seriale del programma che calcola l'evoluzione di un insieme di oscillatori accoppiati. Il programma produce una immagine bidimensionale in cui ogni riga mostra le energie delle n - 1 molle in ogni istante di tempo. Al termine dell'esecuzione il programma dovrebbe produrre un file `coupled-oscillators.ppm` contenente una immagine simile alla seguente:



Parallelizzare la funzione `step()` trasformandola (in tutto o in parte) in un kernel CUDA.