Corso di High Performance Computing

Esercitazione CUDA del 4/5/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/05/05

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 (versione 6.0) comprensivo di compilatore `nvcc`, e alcuni editor 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/ex1-cuda.zip
unzip ex1-cuda.zip
cd ex1-cuda/
```

0. Familiarizzare con l'ambiente

Sulla macchina è installato il compilatore `nvcc` visto a lezione. E' installato anche il programma `deviceQuery` che visualizza le caratteristiche delle GPU a disposizione. Il server `disi-hpc-cuda` dispone di due schede video che supportano CUDA; di default viene utilizzata la prima (Tesla C870), che dispone di un numero maggiore di CUDA core e di una quantità maggiore di RAM rispetto all'altra (GeForce 8600 GT). E' possibile selezionare la scheda da usare sia da programma (es., `cudaSetDevice(1)` usa la GeForce), sia mediante la variabile d'ambiente `CUDA_VISIBLE_DEVICES`; ad esempio

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

esegue il programma `cuda-stencil1d` sulla scheda GeForce, mentre

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

utilizza la Tesla C870 (che è il default se non si specifica altrimenti).

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:

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

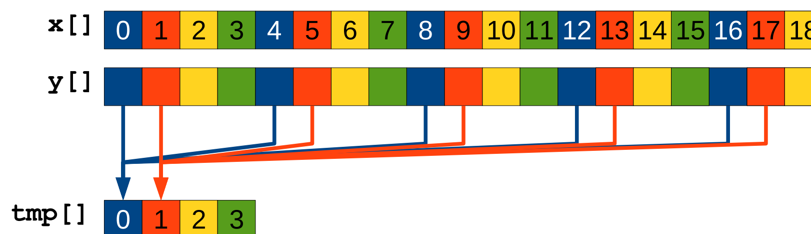
Sono necessarie diverse modifiche alla funzione `dot()` per sfruttare la GPU. Per questo esercizio si richiede di utilizzare *un singolo* blocco composto da *BLKSIZE* threads, procedendo come segue:

1. La CPU alloca sulla GPU un array `tmp[]` di *BLKSIZE* elementi, oltre ad una copia degli array `x[]` e `y[]`. Per allocare l'array `tmp[]` è possibile, a scelta, utilizzare la funzione `cudaMalloc()`, oppure definire `tmp[]` come variabile globale specificando l'attributo `__device__`:
   `__device__ float tmp[BLKSIZE];`
   In questo caso ricordo che per copiare da/verso `tmp[]` occorre usare `cudaMemcpyToSymbol()` e `cudaMemcpyFromSymbol()` al posto di `cudaMemcpy()`.

Inoltre, le variabili definite con attributo `__device__` devono essere globali.

2. La CPU attiva un singolo thread block composto da *BLKSIZE* thread
3. Il thread t (t = 0, … *BLKSIZE* - 1) calcola il valore dell'espressione ($x[t] \times y[t] + x[t + BLKSIZE] \times y[t + BLKSIZE] + x[t + 2 \times BLKSIZE] \times y[t + 2 \times BLKSIZE] + \ldots$) e memorizza il risultato nell'elemento `tmp[t]`.
4. Una volta che il kernel completa l'esecuzione, la CPU calcola la somma dei valori dell'array `tmp[]`, il cui contenuto deve essere prima trasferito nella memoria dell'host. Il risultato della somma è il prodotto scalare.

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 *BLKSIZE* = 4 e n = 19; il thread t memorizza il risultato parziale nell'elemento `tmp[t]`.



Il programma deve funzionare correttamente per qualunque valore di n, che pertanto non deve necessariamente essere un multiplo di *BLKSIZE*.

Attenzione: il programma cuda-dot.cu fa uso di aritmetica in virgola mobile in precisione singola (tipo *float*); si verificano errori dovuti a perdita di accuratezza quando $n > 1.6 \times 10^7$, che potrebbero essere evitati usando *double* al posto di *float*. Purtroppo la GPU presente sul server non supporta l'aritmetica in precisione doppia (quelle più recenti la supportano), quindi è necessario eseguire questo programma con array di lunghezza massima pari a circa 10 milioni di elementi.

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. E' 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 in ordine inverso; il secondo kernel deve invertire gli elementi di `x[]` "*in place*", ossia modificando `x[]` 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.

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 pari o quelli dispari. Una seconda possibilità è di lanciare $n/2$ cuda thread, facendo in modo che tutti i thread siano sempre attivi durante ogni fase e gestiscano ciascuno una coppia di elementi.