Progetto di High Performance Computing 2018/2019

Diego Mazzieri, matr. 0000792583

31/03/2019

Introduzione

Partendo dalla versione seriale fornita, ho creato un modello da usare come base indipendentemente dalla scelta tra OpenMP, MPI o CUDA, in cui ho effettuato delle modifiche volte a semplificare e ottimizzare le soluzioni parallele che avrei dovuto implementare. Ad esempio, ho aggiunto una *ghost area* composta da porzioni di terreno tutte con energia cinetica pari a 0 intorno al dominio preso in considerazione, grazie alla quale è stato possibile evitare, in *propagate_energy*, il controllo sull'indice delle celle adiacenti, essendo sicuro della loro appartenenza alla griglia.
Per facilitare il controllo sulla correttezza delle soluzioni implementate, possibile solo confrontando la similarità delle immagini generate dalle varie versioni del programma, ho deciso di creare uno script bash *check.sh*, grazie al quale questa operazione è stata automatizzata considerando come dimensione della griglia i parametri di input, mentre come numero di steps 100000 (modificabile cambiando la variabile STEPS), analogamente all'immagine fornita come esempio.
Per automatizzare l'esecuzione dei benchmark sulle varie implementazioni, invece, ho creato lo script bash *bench.sh*, che opera in maniera analoga a *check.sh* ma su versioni del programma appositamente pensate per esso, il cui unico output è il tempo di esecuzione. Per riuscire in questo intento senza fornire due versioni (una "normale" e una "benchmark") per ogni implementazione, ho modificato il *Makefile* che ci era stato fornito aggiungendo il target *benchmark* e ho fatto uso nel codice, ove necessario, della macro BENCHMARK, utile a differenziare in fase di compilazione l'output del programma.

Versione OpenMP

Dovendo essere ripetute le stesse operazioni sulla griglia per *nsteps* volte ed essendo tale valore potenzialmente molto grande, ho preferito, analogamente a quanto visto per l'algoritmo *Odd-Even Transposition Sort*, riutilizzare il pool di thread invece che ricrearlo ogni volta come sarebbe successo mediante la direttiva *#pragma omp parallel for*. Per questo motivo, ho usato la direttiva *#pragma omp parallel* per racchiudere il ciclo sugli istanti temporali presente nella funzione *main* e la *#pragma omp for* per parallelizzare i cicli che, nella versione seriale, venivano eseguiti nelle funzioni *increment_energy, count_cells, propagate_energy* e *average_energy*.
Nonostante la presenza della direttiva *#pragma omp parallel*, che potrebbe essere fuorviante, il ciclo sugli istanti temporali non è stato parallelizzato, infatti lo stato di una determinata porzione di terreno in un qualsiasi momento *t* è strettamente dipendente dallo stato della stessa all'istante *t-1*, che deve quindi essere stata calcolata precedentemente.
Essendo il carico di lavoro tra i vari thread molto simile per tutte le parti che sono state parallelizzate, si è deciso di mantenere sempre lo scheduling statico utilizzato di default dal compilatore GCC.

Versione CUDA

Per la versione CUDA del programma, ho come prima cosa trasformato le funzioni *increment_energy, count_cells, propagate_energy* e *average_energy* in kernel. Dovendo questi ritornare forzatamente *void*, per quelli incaricati di effettuare la riduzione, è stato necessario inserire un parametro aggiuntivo in cui memorizzare il risultato. Nel caso di *average_energy*, inoltre, tale parametro, una volta calcolato, deve essere diviso per la dimensione della griglia in modo da ottenere la media dello step corrente. Essendo possibili, all'interno di un kernel, solo meccanismi di sincronizzazione a livello di blocco (*__syncthreads()*), per effettuare tale divisione in sicurezza ho deciso di separare le operazioni in due kernel distinti: *sum_energy* si occupa di sommare le energie cinetiche di tutte le celle del dominio, *average_energy* di dividere tale valore per il numero di celle.
Per inizializzare a 0 le locazioni di memoria contenenti il risultato della riduzione ho fatto uso di *cudaMemset*[1].
Successivamente, ho scelto per ogni kernel la dimensione della griglia e dei blocchi in cui partizionare le varie porzioni di terreno e ho ragionato sui possibili benefici dell'uso della *shared memory*. Nello specifico:

- Per *increment_energy* ho utilizzato una griglia 1D composta da blocchi da 1024 thread, ognuno incaricato di aggiornare la propria cella solo se questa non è ai bordi del dominio o al di fuori di esso;

- Dovendo sia in *count_cells* che in *sum_energy* operare una riduzione su una matrice, ho utilizzato, per semplicità, blocchi 1D da 1024 thread ognuno considerandola come un array. Per effettuare tale operazione su x celle, è necessario leggere e modificare la prima $log_2(x)$ volte, la seconda e la terza $log_2(\frac{x}{2})$ volte, le successive quattro $log_2(\frac{x}{4})$ … quindi ho fatto uso della *shared memory* così da limitare gli accessi alla memoria globale. Per permettere all'algoritmo di funzionare anche quando il numero di celle non fosse stato multiplo della dimensione dei blocchi (1024), ho considerato, nel caso di indice *out of bound*, un valore nullo così da non influire sul risultato della somma e evitare ulteriori controlli;

- Anche nel caso di *propagate_energy*, ho fatto uso della *shared memory* in quanto ogni cella, avendo necessità di controllare l'energia di quelle ad essa adiacenti, deve essere letta da un minimo di 3 fino ad un massimo di 5 volte. A differenza dei kernel precedenti, tuttavia, per agevolare il calcolo dello *stencil*, ho utilizzato un partizionamento 2D sia per la griglia che per i blocchi. Il partizionamento della griglia tiene conto della presenza di *ghost cells* anche nella *shared memory*, analogamente a quanto visto a lezione per la soluzione dell'automa *Anneal*.

Stampare il numero di celle maggiori della soglia e l'energia media ad ogni iterazione avrebbe comportato un collo di bottiglia nel programma dovuto al dover trasferire due soli valori alla volta. Per ovviare a questo problema, ho pensato di allocare simmetricamente nella GPU e nella CPU, due array grandi quanto il numero di iterazioni da effettuare, alle cui posizioni i-esime sono memorizzate rispettivamente le celle sopra la soglia e l'energia media dell'iterazione corrispondente. Questi array vengono trasferiti dalla GPU alla CPU per essere stampati solo una volta terminata la computazione, in modo da sfruttare in maniera più efficiente possibile il buffer a disposizione.

Valutazione prestazioni

La *Figura 1*, riportata di seguito, mostra i grafici dei tempi di esecuzione e del relativo speedup delle tre versioni del programma (Seriale, OpenMP, CUDA) sul server *isi-raptor03.csr.unibo.it* che monta un processore Intel Xeon E5-2603 v4 @ 1.70GHz con 12 core, 64 GB di RAM e tre GPU NVidia GeForce GTX 1070 (di cui ne usiamo solamente una). Come *nsteps* ho considerato il valore predefinito 2048 (variabile STEPS nel file *bench.sh*), mentre come dimensioni *n* della griglia, ho scelto, considerando come passo 100, i valori da 0 a 1000. Per ogni *n* preso in considerazione e per ogni versione del programma, il tempo di esecuzione è stato misurato per 5 volte (variabile TESTS nel file *bench.sh*) ed è poi stata fatta una media.

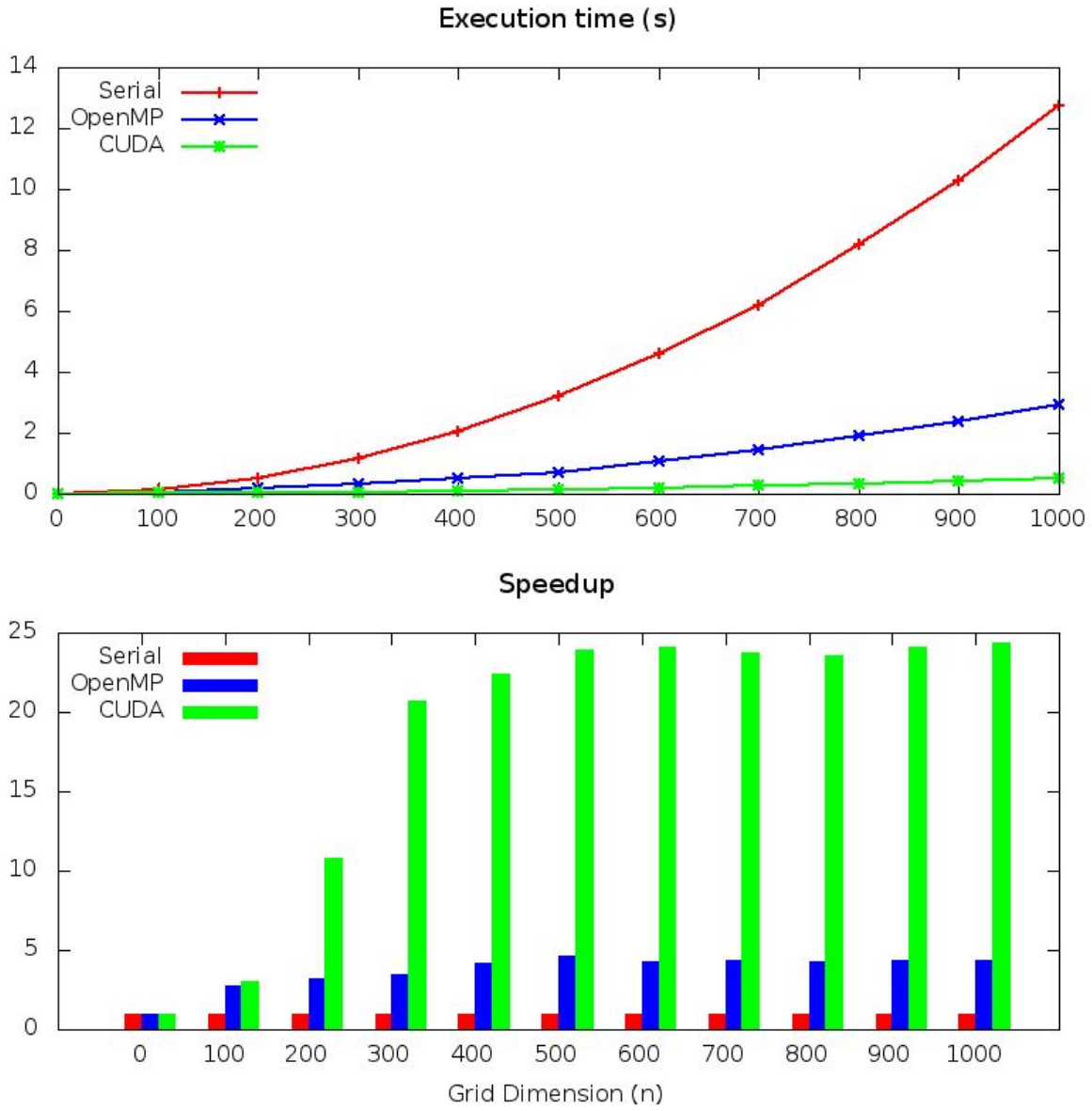

Figura 1: Tempo di esecuzione e Speedup delle implementazioni Seriale, OpenMP e CUDA al variare della dimensione dell'input n, nsteps=2048

Conclusioni

Intenzionalmente, nel benchmark effettuato, non ho tenuto conto né della fase di setup del programma, in cui viene inizializzata serialmente la griglia con valori casuali, né di quella di stampa su terminale dei risultati, ma solo di quella inerente l'evoluzione effettiva dello *stencil*, considerando però, nel caso di CUDA, anche il tempo necessario a trasferire i valori calcolati dalla GPU alla CPU.
Dai risultati ottenuti, come potevamo aspettarci, notiamo come mano a mano che aumenta la dimensione della griglia, sono sempre più evidenti i benefici computazionali dell'utilizzo del parallelismo a memoria condivisa di OpenMP, che raggiunge uno speedup **> 4x** rispetto alla versione seriale per valori di $n > 400$ e del parallelismo massivo di CUDA, che registra sempre valori intorno a **24x** rispetto alla versione seriale quando $n > 500$.

Riferimenti bibliografici

1. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__MEMORY_ge07c97b96efd09abaeb3ca3b5f8da4ee.html#ge07c97b96efd09abaeb3ca3b5f8da4ee