Edoardo Carlini 30/12/2022

RELAZIONE DEL PROGETTO GPU101

SCOPO DEL PROGETTO

Il progetto consiste nel prendere un algoritmo già implementato in C/C++, che opera in modo sequenziale sulla CPU e crearne una versione altamente parallelizzata, in grado di operare sulla GPU in modo avvantaggiarsi dell'elevato numero di core a sua disposizione e offrire prestazioni migliori nonostante la loro frequenza sia notevolmente più bassa. Il nuovo formato del programma sarà in linguaggio CUDA (.cu), una variante del C che utilizza la stessa sintassi.

Tale lavoro sarà svolto creando una apposita funzione, chiamata Kernel function, in grado di utilizzare la logica dei core grafici. Il codice dovrà essere scritto in modo tale che i cicli che normalmente sono sequenziali possano essere eseguiti nella maggior parte dei casi in parallelo o per riduzione binaria a partire da un elevato numero di partenza.

DATI FORNITI

I dati forniti per risolvere il problema sono una matrice sparsa A e un vettore di numeri casuali B, bisogna trovare il vettore incognito X che, moltiplicato con la matrice A, dia il vettore risultante già noto. I vettori B e X avranno la stessa lunghezza pari al numero di righe della matrice A.

La matrice sparsa è un tipo di matrice in cui il numero di zeri è nettamente superiore al numero di valori diversi da zero in essa contenuta. Per rendere l'elaborazione più efficiente la matrice viene rielaborata e rappresentata in formato CSR, compressed sparse row. Tale formato trasforma la matrice in tre vettori per rappresentare i dati eliminando tutti gli zeri in essa contenuti:

- NNZ = numero di elementi diversi da zero contenuti nella matrice;
- V [NNZ] il primo vettore è lungo NNZ e contiene tutti gli elementi diversi da zero presenti all'interno della matrice sparsa e ordinati in modo secondo la lettura sequenziale di essa, riga per riga;
- RIGA [m+1] il secondo vettore è lungo m+1 e codifica gli indici di V e di COLONNA dove la riga data inizia;

- COLONNA [NNZ] il terzo vettore contiene gli indici di riga ed è lungo NNZ, questo è equivalente a indicare il numero totale di elementi diversi da zero presenti alla riga corrispondete RIGA [i], con i che appartiene al seguente intervallo [0 ; NNZ].

ALGORITMO DA IMPLEMENTARE

L'algoritmo che dobbiamo rielaborare è il SYMGS che sta per Symmetric Gauss-Seidel Smoother; in analisi numerica il metodo di Gauss-Seidel è un metodo iterativo, simile al metodo di Jacobi, per la risoluzione di un sistema di equazioni linear, scritto nella forma matriciale $Ax = b$.

L'algoritmo che dovremmo ottimizzare prevede di eseguire operazioni di moltiplicazione di un vettore noto b per una matrice nota A al fine di ricavare un vettore incognito x.

Tale funzione ha lo scopo di trovare una soluzione per convergenza tramite la moltiplicazione di un vettore per una matrice sparsa e ha lo scopo di testare la latenza del processore. Questo algoritmo, tuttavia, non è parallelizzabile e richiede alcune modifiche per poterlo diventare in quanto ogni passo utilizza tutti i K+1 elementi trovati fino al passo K e richiede di poter sovrascrivere l'elemento K+1.

Innanzitutto, moltiplicare una matrice contente un numero di zeri notevolmente maggiore dei numeri diversi da zero rende questa operazione molto onerosa in quanto tutte le volte che moltiplichiamo un valore per zero eseguiamo dei calcoli inutili. Per evitare ciò i dati in ingresso vengono compressi in formato CSR eliminando tutto gli zeri e quindi le operazioni inutili.

ANALISI E SOLUZIONE

Il SYMGS opera in modo sequenziale per approssimazioni successione dei valori trovati fino alla K+1-esima iterazione, tale sistema sfrutta molto bene l'architettura della CPU, la quale dispone di cache e registri molto veloci e operando ad elevate frequenze, in confronto a una GPU, riesce a completare il lavoro in un tempo piuttosto rapido per input limitati. Tuttavia, questo algoritmo è sequenziale e non parallelizzabile, il che lo rende poco efficace su input di grandi dimensioni e non scalabile su sistemi dotati di un grande numero di core, i quali posso parallelizzare tali operazioni.

L’algoritmo di Jacobi risponde esattamente a questa esigenza eliminando i vincoli di sequenzialità e rendendolo quindi completamente parallelizzabile; in questo modo potremmo avvalerci del grande numero di core disponibili su una GPU per poter eseguire il maggior numero di calcoli possibili in parallelo. A differenza del SYMGS Jacobi non ha bisogno degli elementi calcolati fino a quel momento e di doverli sovrascrivere; pertanto, ogni operazione su ogni singolo elemento può essere eseguita in parallelo con le altre riducendo notevolmente i tempi di computazione.

CONCLUSIONI

Per elaborare grosse moli di dati indipendenti tra di loro e che richiedono un numero elevato di passaggi è decisamente più veloce ed efficiente avvantaggiarsi di una GPU quando possibile. Le GPU operano a frequenze notevolmente più basse delle CPU; tuttavia, il numero di core a disposizione è enormemente maggiore, parliamo di miglia di core contro alcune decine al massimo. Un ulteriore vantaggio sta nel fatto che la memoria grafica è montata direttamente sulla scheda grafica aumentando significativamente la velocità di trasferimento e abbattendo la latenza.

ESECUZIONE

Su github oltre al programma stesso in formato cuda è anche già presente la sua versione compilata; pertanto, per eseguirlo non sarà necessario compilarlo. Nel caso lo si volesse comunque compilare, una volta raggiunta la directory in cui è presente il file contente il codice si dovrà lanciare la seguente istruzione:

nvcc symgs-csr.cu -o symgs-csr

e per eseguirlo:

symgs-csr /nomefile

il programma stamperà a schermo i tempi di elaborazione sia di CPU che di GPU