Progetto del corso di GPU101

implementazione CUDA/C++ dell'algoritmo symgs smoother

Francesco Galbiati

Dicembre 2022

1 Introduzione

Il progetto conclusivo del corso di GPU101 consiste nel fornire un'implementazione di un algoritmo assegnato, che sfrutti le capacità di parallelismo di una GPU per portare a termine la computazione.

1.1 L'algoritmo

L'algoritmo che mi è stato assegnato è il symgs smoother (di cui è stata fornita anche un'implementazione sequenziale come punto di partenza) e l'input che utilizza è costituito da un vettore generato casualmente x con dimensione n e da una matrice quadrata sparsa m (di dimensione n x n), letta da un file e memorizzata in formato CSR (Compressed Sparse Row). L'algoritmo si compone di due fasi: il *forward sweep* e il *backward sweep*; nella prima fase la matrice viene scorsa dalla prima all'ultima riga e vengono effettuati i seguenti calcoli:

```
1     for(int i = 0; i < num_rows; i++){
2         float sum = x[i];
3         const int row_start = row_ptr[i];
4         const int row_end = row_ptr[i + 1];
5         float currentDiagonal = matrixDiagonal[i];
6
7         for (int j = row_start; j < row_end; j++)
8         {
9             sum -= values[j] * x[col_ind[j]];
10        }
11
12        sum += x[i] * currentDiagonal;
13
14        x[i] = sum / currentDiagonal;
15    }
```

La seconda fase è analoga alla prima, con l'unica differenza che la matrice è scorsa dall'ultima alla prima riga. Dal codice si nota che l'esecuzione della riga n utilizza potenzialmente i risultati calcolati da tutte le n - 1 righe precedenti, non sarà quindi possibile eseguire tutte le righe in parallelo.

1.2 Mezzi utilizzati

Non disponendo di una macchina con una GPU Nvidia (necessaria se si vuole usare CUDA) ho utilizzato, come consigliato dal docente del corso, Google Colab per compilare ed eseguire il codice.

2 Implementazione con CUDA

Come evidenziato nella sezione precedente, l'esecuzione dei calcoli su tutte le righe contemporaneamente è impossibile, inoltre l'esecuzione di una sola riga alla volta richiederebbe troppo tempo a causa delle dimensioni della matrice, una soluzione più vantaggiosa è quella di identificare all'inizio dell'esecuzione quali tra le righe della matrice non utilizzano nei loro calcoli risultati di righe precedenti, in altri termini identificare quali righe hanno tutti gli elementi non nulli nella matrice triangolare superiore (diagonale inclusa). Una volta individuate le righe "pronte" si possono eseguire i loro calcoli e salvarne i risultati, avendo cura di salvarli in un array separato (in modo da non fornire dati sbagliati alle iterazioni successive). Questa operazione porterà altre righe della matrice, precedentemente non pronte, a poter essere eseguite, in quanto le dipendenze che bloccavano i loro elementi nella matrice diagonale inferiore sono state risolte. L'algoritmo procede quindi iterativamente fino a quando i risultati di tutte le righe sono stati calcolati. Come per l'implementazione sequenziale anche in questo caso il *backward sweep* è analogo alla prima parte, con l'unica differenza che le righe da considerarsi indipendenti sono quelle con solo elementi non nulli nella matrice triangolare inferiore (diagonale inclusa).

Nella mia implementazione mi sono servito di due array *ready* e *done*, usati come flag per segnalare lo stato di esecuzione di ogni riga della matrice, e di quattro funzioni, delle quali segue la descrizione:

2.1 Funzioni *fready* e *bready*

Le funzioni *fready* e *bready* hanno il compito di determinare, data la matrice in formato CSR e gli array di flag *ready* e *done*, se nel momento dell'esecuzione una riga qualsiasi r è o meno pronta per essere eseguita, in altre parole se agli indici di colonna di tutti gli elementi della matrice triangolare inferiore (superiore nel *backward sweep*) correisponde un valore 1 nella corrsipondente posizione di *ready*. In caso affermativo la flag corrispondente alla riga analizzata viene posta a 1, altrimenti viene lasciata a 0 e verrà valutata in un'iterazione futura. L'esecuzione su GPU di questa funzione necessita l'assegnamento di un singolo thread per riga, che effettuerà la valutazione in maniera indipendente dagli altri thread, di conseguenza ho diviso l'esecuzione su $numrows/1024 + 1$ blocchi, ciascuno con 1024 thread.

Segue lo pseudocodice dell'algoritmo di *fready*:

```
1     if(threadID < num rows){
2         if(NOT(done[threadID])){
3             rowl <= row_ptr[threadID]-row_ptr[threadID + 1]
4             ready[threadID] <= 1
5             for(j from 0 to rowl){
6                 if(NOT(done[row_ptr[thredID] + j]) AND
7                     col_ind[row_ptr[threadID] + j]>threadID){
8                     ready[threadID] <= 0
9                 }
10            }
11        }
12    }
```

L'unica differenza nell'algoritmo di *bready* è un $<$ alla riga 7 al posto del $>$. Al termine di questa funzione le righe che potranno essere eseguite avranno la flag *ready* uguale a 1 e la flad *done* uguale a 0.

2.2 Funzioni *fdo* e *bdo*

Una volta determinate quali righe sono pronte per essere eseguite, viene chiamata la funzione *fdo* (*bdo* nel *backward sweep*), il cui scopo è quello di effettuare i calcoli visti nella sezione 1, avendo cura di utilizzare i valori "vecchi" di x per gli elementi appartenenti alla matrice triangolare superiore e quelli "nuovi" per gli elementi appartenenti alla matrice triangolare inferiore. Per questo kernel è necessario più di un thread a riga, ma al tempo stesso le righe devono essere eseguite in maniera indipendente l'una dall'altra, l'esecuizione viene quindi divisa un numero di blocchi pari al numero delle righe della matrice, ciascuno con $2^{(log_2 maxl)+1}$ threads, dove *maxl* è il numero di elementi della riga più popolata. Il codice si serve di un array ausiliario *tmp* per conservare i risultati delle computazioni del *forward sweep* senza dover sovrascrivere l'array x, e della variabile intera c, che conta quante righe sono state risolte fino a quel momento.

```
1     if(NOT(done[row]) AND ready[row]){
2         rowl <= row_ptr[row+1] - row_ptr[row]
3         sum <= 0
4         __syncthreads()
5         if(threadID < rowl){
6             if(col_ind[row_ptr[row]+threadID]<row){
7                 sum <= sum + values[row_ptr[row]+threadIdx.x]*
8                         tmp[col_ind[row_ptr[row]+threadIdx.x]]
9             } else{
10                sum <= sum + values[row_ptr[row]+threadIdx.x]*
11                        x[col_ind[row_ptr[row]+threadIdx.x]]
```

```
12              }
13          }
14          __syncthreads()
15          if(threadID = 0){
16              sum <= x[row] - sum
17              sum <= sum + matrixDiagonal[row] * x[row]
18              tmp[row] <= sum / matrixDiagonal[row]
19              done[row] <= 1
20              c <= c + 1
21          }
22      }
```

Per effettuare la somma degli elementi nella riga è stata preferita una somma su un'unica variabile da parte di tutti i threads a una riduzione parallela, questo perché, considerando la sparsità della matrice, la riduzione parallela non conviene nel caso in cui i valori da sommare siano pochi, se non uno solo, come può spesso accadere in una matrice molto sparsa. Altri aspetti da notare sono che la variabile *sum* è allocata in shared memory, in quanto viene acceduta più volte, e che l'ultima porzione di codice viene eseguita solo da un thread di ciascun blocco, per fare in modo che le operazioni vengano eseguite solo una volta. Come per le altre funzioni la versione per il *backward sweep* è analoga a quella per il *forward sweep* con l'eccezione della riga 6 in cui è presente un $>$ al posto del $<$. Il *backward sweep* inoltre utilizza *tmp* come array da cui prendere i rvalori per i calcoli e x come array in cui salvare i risultati, per cui alla fine dell'esecuzione i risultati si troveranno in x.
Terminata l'esecuzione di questo kernel il programma prosegue con una nuova *fready* (o *bready*), fermandosi solo quando c raggiunge un valore pari al numero di righe, a quel punto l'esecuzione passa al *backward sweep*.

2.3 Host code

La funzione dell'*host code* in questa implementazione è solamente quella di chiamare iterativamente i kernel della funzione ed azzerare le flag tra il *forward sweep* e il *backward sweep*.

```
1       while(c < num_rows){
2           fready<<<>>>(...)
3           fdo<<<>>>(...)
4       }
5       c <= 0
6       done[*] <= 0
7       while(c < num_rows){
8           bready<<<>>>(...)
9           bdo<<<>>>(...)
10      }
```

2.4 Verifica dei risultati

Come detto nella sezione 2.2 i risultati finali si trovano nell'array x, al termine dell'esecuzione essi vengono confrontati con quelli ottenuti dalla CPU, in modo da verificarne la correttezza; tuttavia i diversi modi in cui CPU e GPU gestiscono la notazione floating point porta delle leggere differenze nei risultati delle medesime operazioni matematiche, che tendono ad aumentare in maniera proporzionale rispetto alla grandezza numerica del risultato stesso. Per questo motivo invece di controllare l'effettiva correttezza del risultato tramite l'operatore ==, si è preferito controllare che la differenza tra i risultati presenti allo stesso indice non superasse una determinata soglia, in questo caso 10^{-3}, viene stampata inoltre la percentuale di precisione del calcolo.
Pochi risultati, mediamente 2.8 con un picco di 8 (su 10 misurazioni), superano la soglia, inoltre se vengono controllati i valori ci si accorge che sono numeri molto alti, dell'ordine delle migliaia, e che la differenza tra i due risultati è comunque inferiore a 10^{-2}.